UNIVERSITÀ DEGLI STUDI DI PISA
FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Tesi di Laurea:

# Progetto e Realizzazione di un Kernel Real-Time Modulare

Autore:
Paolo Gai

Relatori:
Prof. Paolo Ancilotti
Prof. Giorgio Carlo Buttazzo
Prof. Cosimo Antonio Prete

Anno Accademico 1998-1999

*ai miei genitori*

*Con questo lavoro si avvera un piccolo sogno, coltivato fin da quando ero bambino. Se veramente la Cultura è ciò che rimane dopo aver dimenticato tutto, allora spero che questa avventura rimanga per sempre vicino a me, indelebile.*

*Desidero ringraziare tutti i sognatori che mi hanno aiutato lungo il cammino, sopportandomi in questi lunghi mesi di lavoro.*

*In particolare Giorgio, Luca, Luigi, Peppe, Gerardo, Marco e Tonino per la loro non comune disponibilità dimostrata, per la loro simpatia e per i loro utili consigli.*

*Desidero inoltre ringraziare Tomas, Marco, Danilo e Claudio per le preziose indicazioni fornitemi, ed infine Paolo per la simpatia e l'amicizia dimostrata durante le numerose pause in Piazza dei Miracoli.*

**Sommario**

Il lavoro compiuto in questa tesi ha come scopo la realizzazione di un Kernel real time modulare compatibile con lo standard POSIX IEEE 1003.13-1998 profilo PSE5 - Realtime Controller System Profile.

Con Kernel real time modulare si intende un sistema capace di supportare molteplici algoritmi di schedulazione, server aperiodici e protocolli di accesso condiviso in modo trasparente per le applicazioni e per i vari moduli e librerie che interagiscono con il sistema. Particolare attenzione è stata posta nella definizione di una interfaccia che permetta ad un sistemista una implementazione chiara ed efficiente di nuovi algoritmi e protocolli; con tale interfaccia sono stati realizzati svariati moduli che implementano i seguenti algoritmi e protocolli: EDF (Earliest Deadline First), RM (Rate Monotonic), RR (Round Robin), l'algoritmo previsto dallo standard POSIX, TBS (Total Bandwidth Server), CBS (Constant Bandwidth Server), PS (Polling Server), DS (Deferrable Server), PI (Priority Inheritance), PC (Priority Ceiling), SRP (Stack Resource Policy), NPP (Non Preemptive Protocol).

Il Kernel risulta inoltre compatibile con un subset dello standard POSIX, in particolare con il profilo PSE52 che prevede un supporto per le estensioni real time dei sistemi POSIX all'interno di un ambiente monoprocesso multithread.

Il Kernel infine dispone di funzionalità che permettono la registrazione di statistiche inerenti i tempi di calcolo dei vari thread all'interno del sistema, predispone le infrastrutture necessarie per l'utilizzo di un tracer realtime e di un filesystem e dispone di librerie per gestire i più comuni device presenti su di un PC, quali ad esempio la tastiera, il mouse, le schede video, di rete e sonore.

# Indice

# Chapter 1

# Introduzione

## 1.1 I sistemi operativi in tempo reale

Lo sviluppo delle tecnologie informatiche ed in particolare la crescente potenza messa a disposizione dai nuovi processori ha allargato notevolmente il numero di applicazioni in cui un calcolatore elettronico può essere efficacemente utilizzato.

Alcune di queste applicazioni (che spaziano dai distemi dedicati a quelli multimediali) sono caratterizzate da requisiti temporali e computazionali. Per questa ragione la ricerca nell'ambito dei sistemi operativi real time si è spostata negli ultimi anni verso problematiche che non comprendono solo una garanzia hard dei task appartenenti alla applicazione, ma includono anche temi collaterali quali la fault-tolerance ed il supporto per computazioni non critiche garantite basandosi su parametri statistici ed approcci adattivi.

Il problema di molti approcci di questo tipo è che spesso essi non sono stati realizzati e testati su applicazioni reali, ma simulati su appositi strumenti di sviluppo per mezzo di carichi sintetici generati con distribuzioni casuali. La mancanza di una implementazione è dovuta spesso alla impossibilità (o comunque alla laboriosità) di adattamento dei sistemi operativi disponibili, ed alla difficoltà di implementare un nuovo Kernel per ogni nuovo algoritmo sviluppato.

Per questo motivo la maggior parte dei sistemi operativi commerciali ha adottato politiche di schedulazione statiche (ossia, a priorità fissa) che purtroppo permettono di implementare solo un numero limitato di politiche, escludendo implicitamente l'apertura dei sistemi alla implementazione di nuovi algoritmi per la gestione di tas e risorse condivise.

## 1.2 Obiettivi del progetto

Per ovviare alle problematiche presentate nella precedente Sezione, il laboratorio RETIS della Scuola Superiore di Studi e Perfezionamento S. Anna ha deciso di avviare lo sviluppo di un nuovo Kernel di ricerca espressamente progettato per fornire un supporto adeguato allo sviluppo di nuovi algoritmi di scheduling per CPU e per le risorse presenti all'interno del sistema.

Particolare cura è stata dedicata alla definizione dei componenti presenti all'interno del sistema e delle loro interfacce reciproche. Il principio fondamentale che ha guidato tale definizione è stato quello di fare si che i vari algoritmi potessero essere implementati in modo indipendente l'uno dall'altro, rendendo in questo modo possibile la loro combinazione per creare configurazioni non previste inizialmente.

Lo sviluppo di un nuovo Kernel non può comunque prescindere da quella che è la realtà applicativa e commerciale degli altri Kernel real time esistenti, pertanto un'altro degli obiettivi del progetto è stato quello di realizzare una interfaccia che fosse il più possibile aderente agli standard industriali, in particolare allo standard POSIX [Soc96] [Soc98]. Per questo motivo sono stati sviluppati dei Moduli che implementano le funzionalità previste dallo Standard, quali ad esempio il supporto per i segnali real-time, per i meccanismi di sincronizzazione (mutex, variabili condition, semafori, timers), per i meccanismi di comunicazione (message queues) e per la gestione dei thread.

Assieme al kernel vengono inoltre fornite delle librerie di supporto che permettono l'interfacciamento delle applicazioni con i device più comuni presenti in un personal computer, quali la tastiera, il mouse, le porte seriali, le schede grafiche, di rete e sonore. Buona parte delle librerie è stata derivata dalla versione precedente del Kernel Hartik, di cui questa tesi è la continuazione.

## 1.3 Descrizione del lavoro

Questo documento intende descrivere le caratteristiche fondamentali che caratterizzano il nuovo Kernel. Particolare spazio verrà dato alla descrizione della architettura generale del sistema, nonchè alle interfacce tra i vari Moduli. Verranno inoltre forniti esempi di implementazione facendo riferimento esplicito al codice che implementa il kernel, senza comunque riportarlo integralmente, data la sua dimensione.

Nel Capitolo 2 verranno indicati utili riferimenti allo stato dell'arte per quanto riguarda i kernel modulari in grado di supportare vari algoritmi di scheduling.

I Capitoli 3 e 4 descrivono l'architettura del sistema; in particolare il Capitolo 3 descrive in generale l'Architettura del Kernel evidenziandone le componenti principali ed i loro compiti, mentre il Capitolo 4 descrive con maggior dettaglio l'architettura utilizzata per implementare la modularità degli algoritmi di scheduling e dei protocolli di accesso a risorse condivise.

I Capitoli 5, 6 e 7 contengono riferimemnti diretti all'interfaccia esportata dai vari componenti del sistema necessari allo sviluppo dei Moduli, evidenziando i tipi ed i parametri, nonchè il posizionamento all'interno del codice sorgente delle principali funzioni del sistema; in particolare il Capitolo 5 descrive l'interfaccia esportata dalla OS Lib; il Capitolo 6 descrive i Modelli e le strutture dati ad essi correlate; il Capitolo 7 contiene la descrizione dei supporti forniti dal Kernel generico per lo sviluppo dei Moduli.

I Capitoli 8 e 9 descrivono in modo ampio l'interfaccia dei moduli di schedulazione e dei moduli che implementano la gestione delle risorse, mentre il Capitolo 10 contiene il riferimento alle varie primitive e funzioni interne del Kernel Generico.

Il Capitolo 11 contiene una serie di esempi che mostrano la struttura di alcuni Moduli implementati. Infine il Capitolo 12 contiene informazioni relative alla compatibilità del sistema con lo standard POSIX.

# Chapter 2

# Stato dell'arte

La letteratura riguardante i sistemi real time ha prodotto negli ultimi anni un grande numero di algoritmi di scheduling, a partire dai classici algoritmi proposti da Liu e layland [LL73], quali Rate Monotonic (RM) per quanto riguarda le priorità statiche ed Earliest Deadline First (EDF) per quanto riguarda le priorità dinamiche.

Spesso l'analisi di schedulabilità per gli algoritmi proposti viene sviluppata sotto ipotesi molto stringenti, quali ad esempio il fatto che i task debbano essere periodici con tempi di calcolo costanti, nonchè il fatto che i Kernel possano schedulare in modo completamente preemptive, senza considerare quindi molte situazioni comuni, quali l'utilizzo di risorse condivise, interruzioni, ecc.

Per questa ragione, molti lavori successivi hanno migliorato l'analisi iniziale proponendo nuovi algoritmi per gestire, ad esempio, richieste aperiodiche e risorse condivise.

Dall'altra parte i sistemi operativi realtime commerciali come QNX [SSL], VX Works [Riv], tendono a minimizzare le sezioni non preemptive del kernel per far tendere lo scheduling reale a quello totalmente preemptive ipotizzato da Liu e Layland, a ridurre i tempi di cambiamento di contesto e quelli di gestione degli interrupt, nonchè ad aumentare l'efficienza generale del kernel.

L'approccio di questi sistemi alla schedulazione rimane comunque simile a quello proposto nello standard POSIX [Soc96] [Soc98], in quanto tali Kernel continuano a basarsi su priorità fisse, rendendo possibile una implementazione semplice dei soli algoritmi a priorità fisse, quali Rate Monotonic ed i suoi derivati.

Per quanto riguarda i kernel di ricerca, RT-Mach [TNR90] [Nak98] costituisce un importante sistema che garantisce delle prestazioni real time implementando direttamente algoritmi di scheduling quali RM, EDF, Priority Inheritance e le CPU reservations [RJMO98], presentando così un ampio spettro, seppur non ampliabile, di algoritmi di scheduling.

Un altro sistema di ricerca sviluppato per fornire supporti per applicazioni real time è Nemesis [LMB$^+$]. Tale sistema è basato su di una struttura verticale non convenzionale. Mentre RT-Mach è un $\mu$kernel il cui obiettivo principale è quello di minimizzare le sezioni non preemptive riducendo la dimensione del Kernel ed implementando le funzionalità del sistema operativo all'interno di processi esterni (chiamati server), Nemesis cerca di implementare tutte le funzioni di sistema operativo all'interno del codice applicativo. Il kernel fornito da Nemesis implementa solo alcune funzionalità utili per accedere all'hardware, fornendo una protezione temporale basata su un algoritmo di allocazione basato su EDF. Se una applicazione necessita di un particolare algoritmo di scheduling tale algoritmo dovrà essere implementato a livello utente.

Esiste poi il Kernel YARTOS [KSD91], in cui gli autori propongono un $\mu$Kernel Real-Time sperimentale con garanzia sui tempi di risposta dei task. Il lavoro si basa su un modello formale sottostante basato su task sporadici con tempi di risposta da garantire e risorse software da condividere.

Un altro approccio esplorato in letteratura è, oltre a scrivere ex-novo di nuovi Kernel, quella di apportare modifiche a kernel convenzionali per adattarli alle richieste dei sistemi operativi real time. Alcuni dei lavori che vanno in tal senso propongono modifiche ai Kernel tradizionali per introdurre appositi punti di preemption, per schedulare autonomamente la gestione degli interrupt o per implementare qualche forma di scheduling dei device.

Generalmente questi approcci introducono nuovi algoritmi di scheduling, ma rimangono comunque di difficile implementazione in quanto molti Kernel tradizionali sono basati su un metodo di allocazione per tick, che permette di implementare solo un limitato numero di algoritmi. Algoritmi come Proportional Share [AB99] [JSMA98] [SAWJ97] [WW94] [SAWJ$^{+}$96], basati su una allocazione della CPU per tick, sono espressamente pensati per essere implementati su di un Kernel tradizionale.

Una recente soluzione è quella proposta nel Kernel RED-Linux [WL99]. Con questo approccio gli autori propongono una metodologia che consente di sviluppare algoritmi di scheduling modulari all'interno di Linux. La metodologia distingue un dispatcher (in pratica uno schedulatore che schedula i propri task basandosi su di una serie di Job presenti in una coda interna) ed un allocatore (che in pratica è uno schedulatore che a livello utente si occupa di fornire i task al dispatcher).

Un altro approccio diverso è quello seguito nel progetto DSRT [hCN99]. Gli autori gestiscono la schedulazione di task con tempo di esecuzione variabile o costante utilizzando un processo ad alta priorità come schedulatore che pone il task da schedulare nel livello di priorità subito inferiore, bloccandosi subito dopo. Questo approccio permette di realizzare all'interno di un normale sistema operativo a priorità fisse, quale Unix, uno scheduler che implementi varie politiche per task di tipo soft, con tutti i vantaggi di configurabilità e gli svantaggi di inefficienza dati dal fatto di avere comunque a che fare con un task utente normale.

Buona parte dei restanti algoritmi di scheduling proposti dalla ricerca sono stati testati e sviluppati con l'ausilio di simulatori dedicati, che permettono di simulare la schedulazione di task set con parametri casuali estratti da distribuzioni desiderate [CBLL98].

In ogni caso, la flessibilità di specifica di nuovi algoritmi di scheduling è divenuta da tempo un argomento importante all'interno della ricerca sui sistemi operativi real-time. Negli ultimi tempi, una serie di Kernel sperimentali stanno iniziando a fornire supporti per implementare differenti algoritmi di scheduling, separando i concetti di algoritmo di scheduling e di meccanismo che permette l'implementazione di tali algoritmi (tale concetto non è comunque nuovo). Ad esempio, il $\mu$kernel L4 [Lie95] [LES$^+$97] fornisce un meccanismo basato sui *preempter threads* per implementare uno scheduling flessibile (anche se il meccanismo di base rimane un meccanismo a priorità fisse). Una soluzione diversa è stata proposta con l'ExoKernel [EK95] [EKO95], che non introduce nessuna astrazione ma lascia al Sistema Operativo il compito di specificare gli algoritmi di schedulazione effettivi.

Un approccio elegante è infine il CPU Inheritance scheduling [FS96], in cui gli autori implementano un nuovo metodo alla schedulazione basato sull'idea che un task può eseguire solo se riceve del tempo macchina da un altro task che ne possiede. Questo approccio permette di realizzare una struttura gerarchica di schedulatori, in cui ciascuno schedulatore riceve del tempo macchina e lo distribuisce ai task che gestisce secondo la propria politica di scheduling. I vantaggi di questo approccio sono una semplice implementazione del meccanismo di priority inheritance (il task bloccato dona risorse ad un altro task) ed il fatto che lo scheduler e' spezzettato in tanti task in modo gerarchico, permettendo la implementazione di scheduler molto flessibili anche all'interno delle applicazioni. Inoltre l'approccio utilizzato si adatta bene ai multiprocessori, nonchè alla implementazione di scheduler processor affinity[1] ed alla gestione di più applicazioni sullo stesso sistema di elaborazione. Il maggior svantaggio di questo approccio è fondamentalmente il maggiore overhead nello scheduling a causa dello spezzettamento dello schedulatore su più task utente.

[1]Uno scheduler processor affinity è uno scheduler che sceglie i task da eseguire anche in base alla situazione della memoria cache al momento dello scheduling.

# Capitolo 3

# Schema generale e funzionalità offerte dal kernel

## 3.1 La storia passata

Lo sviluppo dell'architettura della nuova versione del kernel è stata influenzata da quella delle versioni precedenti dello stesso kernel sviluppate presso il Laboratorio RETIS della Scuola S. Anna.

Per questo motivo vengono qui richiamate (cfr. Figura 3.1) brevemente le caratteristiche salienti della struttura logica dell'ultima versione stabile del vecchio kernel.

Tutti i dettagli architetturali della macchina su cui effettivamente il kernel viene eseguito sono mascherati per mezzo di un opportuno strato software, denominato *Virtual Machine*. Tale strato rappresenta l'astrazione di una macchina generica in grado di offrire un meccanismo di cambio di contesto su cui strutturare la multiprogrammazione, un gestore della memoria e una gestione delle interruzioni hardware.

Il livello successivo è rappresentato dal *Kernel*, contenente le varie primitive di sistema ed il supporto per i vari meccanismi di comunicazione e garanzia.

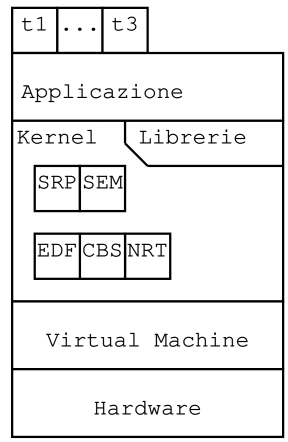


Figura 3.1: Architettura del Kernel Hartik versione 3.3

Dentro tale livello sono implementati, inoltre, gli algoritmi di schedulazione, nella fattispecie:

- Uno scheduler EDF (Earliest Deadline First) [LL73], che permette di gestire task hard periodici e sporadici, con garanzia basata su un approccio a fattore di utilizzazione.
- Uno scheduler CBS (Constant Bandwidth Server) [AB98], che fornisce un supporto per task soft periodici ed aperiodici.
- Uno scheduler prioritario che utilizza l'idle time lasciato libero dai due livelli superiori, utilizzato per gestire i task non real-time (NRT).

I livelli superiori sono rappresentati da *Librerie* che¸ appoggiandosi sul Kernel e sulla Virtual Machine¸permettono di implementare meccanismi di sincronizzazione e comunicazione tra task (semafori, porte, CAB), nonchè i device drivers (tastiera, grafica, suono, scheda ethernet).

Infine l'*Applicazione* vera e propria utilizza i livelli citati precedentemente.

Il Kernel non dispone di un File System e non implementa meccanismi di protezione della memoria.

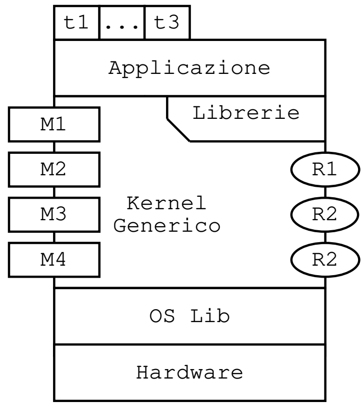


Figura 3.2: Architettura della nuova versione del Kernel

L'intera applicazione viene compilata insieme alle librerie del nucleo che, una volta linkate producono un unico file oggetto caricabile in memoria tramite un Extender proprietario.

## 3.2 Architettura della nuova versione

L'Architettura della nuova versione si inspira alla architettura della versione precedente del kernel. In effetti, ne vengono mantenuti molti approcci nonchè vari aspetti architetturali, come si può vedere dalla Figura 3.2.

In particolare, subito sopra l'hardware viene piazzato uno strato software denominato *OS Lib*, che rappresenta, in modo simile alla Virtual Machine della versione precedente, una astrazione di una macchina generica in grado di offrire un meccanismo di cambio di contesto su cui strutturare la multiprogrammazione, una gestione del tempo e una gestione delle interruzioni hardware orientata agli eventi, ed una parte della libreria C indipendente dal Kernel.

Il livello successivo è costituito da un *Kernel Generico*, che costituisce un insieme di funzioni che implementano le primitive con cui il Kernel stesso si interfaccia con le Applicazioni utente. All'interno di tali primitive non vengono dettagliati gli aspetti di un algoritmo di schedulazione specifico; tali aspetti vengono invece realizzati all'interno di moduli esterni (visibili in figura sotto il nome di *Moduli di Schedulazione* e *Moduli di Gestione delle Risorse*).

Al di sopra del Kernel Generico si posizionano le *Librerie*. Tali librerie implementano il supporto per i dispositivi Hardware utilizzabili nel sistema (tastiera, scheda sonora, scheda grafica¸scheda di rete¸ ecc...), nonchè uno strato di compatibilitá a livello di interfaccia con lo standard IEEE POSIX 1003.13 PSE51 ed alcuni meccanismi di comunicazione (porte, CAB, ecc...).

Come ultimo livello è infine presente l'Applicazione utente.

Di seguito verranno descritte le caratteristiche salienti dell'Architettura. Per una descrizione dettagliata del funzionamento dei moduli e del Kernel Generico si rimanda al Capitolo 4. I successivi capitoli contengono inoltre una definizione ed una descrizione di ciascuna singola funzione di interfaccia implementata.

## 3.3 Modelli

Uno dei concetti fondamentali introdotti con la nuova versione del kernel è il concetto di *Modello*. Con Modello si indica l'insieme delle caratteristiche salienti che identificano la qualità di un servizio fornito o da fornire ad un particolare oggetto del sistema.

Esistono varie tipologie di modelli all'interno del sistema (vedi Tab. 3.1):

**Modelli di Task** Questi modelli vengono utilizzati al momento della creazione di un task ed indicano la Qualità del Servizio che tali task richiedono al sistema per quanto riguarda la *schedulazione* del task in questione. Tali modelli contengono una serie di parametri "comuni"

| Modelli | Momento di utilizzo | Esempi di attributi |
|---|---|---|
| Modelli di Task | Creazione di un task | Deadline¸ WCET¸ Periodo |
| Modelli di Risorse | Creazione di un task | Ceiling di un task, Banda di un File System |
| Modelli di Mutex | Creazione di un mutex | Protocollo da utilizzare (PI, PC, SRP, ...) |

Tabella 3.1: Possibili tipologie di modelli

indipendenti da un particolare algoritmo di scheduling (quali, ad esempio, la dimensione dello stack del task) ed una serie di parametri che indicano vere e proprie "richieste" fatte dal task da creare (quali ad esempio periodo, deadline, ecc...). Notare come la seconda serie di parametri tende ad astrarre anch'essa dall'algoritmo di schedulazione (il concetto di periodo o di deadline è indipendente, ad esempio, dal fatto che venga utilizzato un algoritmo di tipo EDF, RM, CBS od altri...). La funzione di questi modelli è simile a quella delle strutture `pthread_attr_t` definite nello standard POSIX.

**Modelli di Risorse** Questi modelli vengono utilizzati al momento della creazione di un task per descrivere aspetti di Qualità del Servizio legati alle risorse comuni utilizzate dai task. Ad esempio, un task può utilizzare un file system che implementa particolari strategie per la schedulazione delle richieste di accesso ai dati. I parametri necessari per utilizzare tali strategie possono venire passati ai moduli che implementano il file system tramite modelli di risorse. Un altro esempio è dato dai moduli che implementano protocolli di accesso a risorse condivise quali Priority Ceiling e Stack Resource Policy. Questi protocolli richiedono la specifica, per ciascun task che utilizza un protocollo, di parametri quali la priorità di un task o il suo livello di preemption. Ancora una volta, tali parametri saranno forniti al sistema utilizzando un modello di risorsa. La specifica dei modelli di risorse all'atto della creazione di un task è facoltativa.

**Modelli di Mutex** Questi modelli vengono utilizzati al momento della creazione di un semaforo di mutua esclusione per specificare, in modo analogo a quanto fatto per i modelli di task, il protocollo di accesso che il mutex deve utilizzare. La funzione di questi modelli è simile a quella delle strutture `pthread_mutexattr_t` definite nello standard POSIX.

è utile far notare come l'utilizzo dei modelli comporta i seguenti vantaggi all'interno del sistema:

- permette di formalizzare una serie di attributi necessari e sufficienti per caratterizzare una determinata qualità del servizio richiesta da una entità del sistema.

- rende le applicazioni (che utilizzano i modelli) **indipendenti** dai moduli di sistema che realmente implementano e forniscono la qualità del servizio. Ad esempio, la creazione di un task utilizzando un modello specifico (ad esempio, un modello Hard Periodico) prescinde dall'effettivo algoritmo utilizzato per schedulare quel task (per schedulare un task di quel tipo esistono moltissimi algoritmi - ad esempio¸ EDF e RM). Notare che questo permette all'applicazione di essere compatibile con vari algoritmi di scheduling, e pertanto una applicazione di questo tipo continuerà a funzionare se eseguita in ambienti con diversi algoritmi di scheduling (vedi Sezione 3.8). Come effetto collaterale si ha anche che l'utente, al momento di scrivere applicazioni o librerie, può, nei limiti del possibile, prescindere dall'algoritmo di scheduling reale¸ preoccupandosi solo della qualità del servizio che un task deve ricevere.

## 3.4 Applicazione

Una Applicazione all'interno del nuovo Kernel è un insieme di task[1] cooperanti che condividono uno spazio di memoria comune. Non esiste protezione della memoria all'interno del kernel, pertanto nulla viene fatto per evitare che un task utilizzi locazioni di memoria ad esso "in teoria" inaccessibili.

Il paradigma di programmazione supportato dal Kernel è di tipo a memoria comune. La comunicazione tra task pertanto avviene utilizzando memoria condivisa ed opportune primitive di sincronizzazione (il kernel mette a disposizione vari meccanismi di sincronizzazione, quali semafori, mutex, variabili condition, CAB e code di messaggi).

Ogni task è individuato da un Modello di Task, eventuali Modelli di Risorse (se presenti) e da un corpo (body). Sintatticamente il corpo di un task non è altro che una funzione C dichiarata con il seguente prototipo

```
void *body(void *arg);
```

Le Applicazioni possono utilizzare per il loro funzionamento le seguenti funzioni:

- Alcune funzioni della OS Lib utili per accedere agli spazi di I/O;
- Le primitive esportate dal Kernel Generico;
- Eventuali funzioni aggiuntive presenti all'interno dei moduli registrati;
- Le funzioni esportate da librerie, device drivers o dalla libreria standard del linguaggio C;

[1] La letteratura dei Sistemi Operativi usualmente distingue i concetti di *processo* e *thread*. In questo testo con il termine *processo* verrà indicato uno spazio di indirizzi, con uno o più *thread* che eseguono il proprio codice all'interno di tale spazio, e l'insieme delle risorse di sistema allocate per tali *thread*. Con il termine *thread* verrà indicato un singolo flusso di esecuzione all'interno di un processo. Ogni *thread* é identificato da un identificatore univoco e da una serie di informazioni specifiche. Il termine *task* sarà utilizzato come sinonimo del termine *thread*.

- Il sottoinsieme delle primitive POSIX implementate se l'applicazione è una applicazione POSIX.

## 3.5 Moduli

All'interno di questo documento verrà spesso utilizzato il termine *Modulo*. In questo paragrafo verrà spiegato cosa si intende e cosa si può fare con i Moduli.

Un Modulo in generale e' un insieme di dati e operazioni utilizzati all'interno del kernel per implementare una caratteristica del sistema (ad esempio, un algoritmo di scheduling, un gestore delle risorse, ecc...).

Ogni singolo modulo deve in generale *isolare* la propria implementazione rendendola il più possibile indipendente dagli altri moduli presenti nel sistema. In questo modo diventa possibile "montare" vari moduli in modo indipendente, un po' come si fa con i mattoncini di famosi giochi di costruzione, senza che un particolare modulo sia a conoscenza dell'esistenza degli altri suoi simili. Ad esempio, nei limiti del possibile un modulo che implementa un server aperiodico come il Polling Server deve poter funzionare, senza modifiche, appoggiandosi ad un algoritmo di schedulazione statico (quale RM) o dinamico (quale EDF).

Ogni modulo possiede una serie di identificativi simbolici e numerici, che permettono di implementare, direttamente nel modulo, dei controlli di consistenza e compatibilità (ad esempio, si può imporre che un modulo che implementi un algoritmo tipo Total Bandwidth Server non funzioni con uno scheduler di base RM).

Le tipologie di moduli presenti nel sistema sono attualmente tre (tale numero è destinato ad aumentare con l'implementazione di altre parti del sistema, quali il tracer ed il file system), e sono:

- Moduli che implementano algoritmi di schedulazione e server aperiodici;

- Moduli che implementano un gestore di risorse;
- Moduli che implementano protocolli di accesso a risorse condivise (casi particolari dei moduli di gestione delle risorse);

Ciascun tipo di Modulo possiede una interfaccia ben definita e limitata. Tale interfaccia si occupa di gestire tutti gli eventi riguardanti le entità manipolate dal Modulo stesso.

Le funzioni di interfaccia vengono richiamate dal Kernel Generico o da altri Moduli in seguito ad eventi temporali da parte della OS Lib o alla chiamata di primitive di sistema da parte del task in esecuzione.

I moduli utilizzano i Modelli (vedi Sezione 3.3) per "contrattare" i propri servizi con le restanti parti del sistema. Esistono quindi funzioni di interfaccia per gestire i Modelli e per sapere se un un modulo possa o meno gestire una richiesta particolare (ad esempio, un modulo che implementa l'algoritmo di schedulazione EDF può accettare come modelli dei task periodici hard ma non dei task aperiodici).

## 3.6 OS Lib

La OS Lib costituisce uno strato software piazzato subito sopra lo strato Hartdware e permette, esportando una interfaccia ben nota, di nascondere agli strati superiori i dettagli implementativi legati alla architettura presente nel sistema.

La OS Lib e' costituita da due parti. La prima parte permette di eseguire semplici applicazioni monoprogrammate, ed in particolare comprende:

- Un Extender in memoria lineare compatibile con le specifiche multiboot (supportate ad esempio da bootloader free come GRUB). Una applicazione prodotta utilizzando la OS Lib è costituita da un unico file oggetto compatibile con tali specifiche;

- Una libreria di programmazione in modo protetto per le CPU compatibili Intel X86;

- Un sottoinsieme della LIBC standard indipendente dal S.O.

La seconda parte della OS Lib rappresenta un insieme di supporti al kernel, comprendenti una serie di operazioni di basso livello (bootstrap, cambi di contesto hardware, gestione delle interruzioni) e di funzioni utili per implementare la protezione temporale e spaziale dei task (gestione di eventi temporali e della memoria libera).

La OS Lib esporta una visione del tempo macchina che ha le seguenti caratteristiche peculiari.

- Il tempo visibile alle applicazioni è un tempo continuo, monotono crescente, non quantizzato in slice (come di solito avviene in molti sistemi operativi realtime).

- La precisione reale del timer viene nascosta alle applicazioni, che possono cosí utilizzare un orologio virtuale con una precisione del nanosecondo (come del resto stabilito dallo standard POSIX per la struttura `timespec`); la precisione reale del realtime clock è di qualche microsecondo, variabile a seconda dell'hardware su cui viene eseguita l'applicazione.

- I Moduli ed il Kernel Generico possono richiedere alla OS Lib di eseguire degli eventi ad un tempo ben preciso. Un evento è costituito da una normale funzione C, eseguita ad interruzioni disabilitate. La OS Lib garantisce che un evento, una volta creato, non scatterà prima del tempo specificato. In ogni caso sono possibili (ed inevitabili) dei ritardi dell'ordine di qualche decina di microsecondi nel tempo di esecuzione dell'evento rispetto al suo tempo "nominale".

## 3.7 Kernel Generico

Il *Kernel Generico* implementa le primitive con cui il Kernel si interfaccia con le Applicazioni utente e con le Librerie.

Il termine Generico è stato utilizzato in quanto il Kernel è stato sviluppato prescindendo dal significato e dall'algoritmo di scheduling da implementare.

Questo significa che nel Kernel Generico non sono presenti i supporti per i concetti come priorità, deadline, code di task, ecc. L'intero codice del Kernel Generico prescinde dalla implementazione di uno specifico algoritmo di schedulazione; gli aspetti di un algoritmo vengono invece interamente incapsulati all'interno dei Moduli, con cui il Kernel Generico colloquia.

In generale, quindi, l'Applicazione comunica con il kernel tramite delle primitive generiche, richiedendogli l'esecuzione di un task set. Il kernel gestisce i task in funzione dei loro modelli utilizzando i moduli registrati. Pertanto una modifica ai moduli che non influenzi i modelli dei task non comporta modifiche nel kernel e nelle applicazioni.

Il Kernel Generico distingue inoltre i task in due grandi categorie: task di sistema e task utente. I task di sistema sono tipicamente utilizzati dai drivers (come tastiera, mouse, file system, ecc...) per implementare i propri processi esterni. I task utente sono i task che di solito fanno parte della Applicazione vera e propria.

Infine, il Kernel Generico mantiene distinte le attività di scheduling e dispatching:

**scheduling** è l'attività con cui il Kernel richiede ad un modulo l'indicazione di un task da eseguire;

**dispatching** è l'attività con cui il Kernel ordina ad un modulo l'esecuzione di un suo task; normalmente e' il task indicato dallo scheduling, ma non sempre (vedi Sezione 4.4.5).

## 3.8 Inizializzazione e Terminazione del Sistema

Prima che una Applicazione possa essere eseguita occorre in qualche modo informare il Kernel Generico di quali siano i moduli effettivamente richiesti dal codice applicativo, e di come essi vadano configurati.

Per fare questo, al momento di compilare il file oggetto multiboot compliant occorre linkare una apposita funzione di inizializzazione, all'interno della quale vengono "registrati" ed inizializzati i moduli richiesti.

La registrazione di solito comporta anche la creazione di un primo task utente non realtime, task che di solito effettua la chiamata della funzione `main()`, secondo quanto specificato dallo standard C. Oltre a chiamare tale funzione, il primo task tipicamente inizializzerà parte dei device drivers (come, ad esempio, la tastiera).

La terminazione del sistema avviene in seguito alla chiamata di apposite primitive (quali `sys_end()` o `sys_abort()`) oppure alla terminazione dell'ultimo task utente. Il sistema al momento della chiusura tenta di effettuare una chiusura "pulita", cercando di far terminare in modo corretto i rimanenti task di sistema.

Per informazioni più dettagliate fare riferimento alla Sezione 10.2.

# Capitolo 4

# Architettura nel dettaglio

In questo capitolo verranno descritte con maggior dettaglio le architetture dei Moduli di Schedulazione, dei Moduli di Gestione delle Risorse e dei moduli che implementano i protocolli di accesso a risorse condivise tramite mutex. Verranno inoltre descritti alcuni approcci utilizzati nell'implementazione del Kernel Generico.

## 4.1 Kernel Generico

L'esigenza di ottenere la modularità ha portato ad effettuare alcune modifiche nel Kernel Generico che tendono a differenziarlo da un approccio classico allo sviluppo dei Kernel.

Tali modifiche riguardano essenzialmente i seguenti aspetti:

- Struttura del descrittore di task;
- Stato di un task.

### 4.1.1 Descrittore di task

Per implementare un Kernel realmente modulare non è possibile fare affidamento su un descrittore di task monolitico, cioè su un descrittore di task che comprenda al suo interno tutti i possibili campi utilizzabili da un generico Modulo di Schedulazione. Questo perchè, al momento della scrittura del Kernel Generico, non si ha la conoscenza di tutti i Moduli di Schedulazione e Gestione di Risorse che verranno implementati. L'idea di base è che il Kernel Generico *non debba in alcun modo essere modificato* per implementare un nuovo algoritmo.

Per questo motivo il descrittore di task è stato spezzato in più parti: una parte generica, comune ai vari moduli, e una serie di parti dipendenti dai moduli effettivamente registrati al momento della inizializzazione. Le parti di descrittore dipendenti dai moduli sono locali ai moduli stessi.

La parte generica del descrittore di task contiene tutte le informazioni che sono utilizzate dal Kernel Generico per fornire il supporto alle primitive, quali contesto, stack, stato, codice di errore (`errno`), informazioni addizionali sulle statistiche dei tempi di calcolo, ecc.

La parte generica del descrittore non contiene alcuna informazione riguardanti deadline assolute e relative, priorità[1], timeslices, tempi di riattivazione, ecc. Tali informazioni, necessarie per ogni algoritmo di schedulazione che si rispetti, dovranno essere mantenute dal modulo stesso in una zona di memoria ad esso privata. Questo fatto comporta che:

- Il Kernel Generico non si occupa minimamente di gestire concetti riguardanti le politiche di schedulazione.

- Ogni istanza dello stesso modulo ha una propria copia della parte privata in cui registra le proprie informazioni riguardanti i task da esso posseduti (o in esso inseriti da parte di altri moduli, vedi Sezione 4.2).

- Un modulo del sistema può utilizzare solo le informazioni contenute nel descrittore di task generico e le proprie informazioni private, ma non le

[1] In effetti la reale implementazione del descrittore di task comprende un campo contenete una priorità, utilizzato essenzialmente per rendere più semplice e veloce la scrittura dei moduli (vedi 7.2.1).

| Nome Stato | Descrizione |
|---|---|
| FREE | Descrittore di task non utilizzato |
| SLEEP | Stato di default in cui si trova un task subito dopo la creazione; è anche il tipico stato in cui si ritorna dopo una chiamata alla primitiva task_sleep |
| EXE | Stato del task attualmente in esecuzione |
| WAIT_* | Serie di stati in cui si trovano i task in seguito al blocaggio su una primitiva di sincronizzazione |

Tabella 4.1: Stati definiti dal Kernel generico

informazioni private degli altri moduli (a meno che i moduli stessi non siano stati scritti in modo da "conoscersi" e condividere informazioni).

- Tipicamente, un modulo che implementa un Server Aperiodico non accede direttamente alle strutture interne del Modulo di Schedulazione a cui si appoggia, ma utilizza la interfaccia messa a disposizione nel modo opportuno (vedi Sezione 4.2.4).

### 4.1.2 Stato di un task

L'implementazione dei Moduli, ed il conseguente incapsulamento dei dettagli implementativi degli algoritmi all'interno di tali moduli, comporta che il concetto di *stato* di un task non ha più una connotazione "globale" per il sistema, ma una connotazione "locale".

Per connotazione "locale" di uno stato si intende il fatto che il Kernel Generico definisce solo un sottoinsieme degli stati in cui un task si può trovare, lasciando libertà ai moduli di definirsene un sottoinsieme privato.

In particolare, il Kernel Generico definisce solo gli stati descritti in Tabella 4.1. Come si può notare, con i soli stati definiti dal Kernel Generico non è possibile tracciare un diagramma completo delle transizioni di stato dei task all'interno del sistema.

In effetti, gli stati elencati in Tabella 4.1 devono essere integrati con gli

| Nome Stato | Descrizione |
|---|---|
| READY | Questo stato è il classico stato pronto per un task attivo in attesa di essere schedulato. |
| BLOCKED | Questo è lo stato in cui viene posto un task in seguito ad un bloccaggio su un semaforo. |
| ZOMBIE | Questo è lo stato in cui viene messo un task periodico Hard in seguito alla sua terminazione, in attesa che la sua banda venga rilasciata a fine periodo. |
| LOBBY | Alcune implementazioni del protocollo SRP utilizzano questo stato per ritardare l'attivazione di un task in attesa che il ceiling di sistema diventi nullo. |
| IDLE | Questo è il tipico stato in cui viene posto un task periodico alla fine di una istanza, in attesa della riattivazione che avverrà a fine periodo. |
| DELAY | Questo stato viene utilizzato per task in attesa di un timeout. |
| SP_WAIT | Questo stato è utilizzato quando un task sporadico finisce una attivazione e rimane in attesa della fine del minimo tempo di interarrivo per poter accettare nuove attivazioni esplicite. |

Tabella 4.2: Esempi di stati definiti all'interno dei Moduli.

stati interni che andranno definiti all'interno dei Moduli di Schedulazione o di Gestione delle Risorse. Per esempio, all'interno della Tabella 4.2 sono elencati alcuni stati interni utilizzati all'interno dei moduli implementati nel Kernel.

## 4.2 Moduli di Schedulazione

### 4.2.1 Obiettivi dei Moduli di Schedulazione

Un Modulo di Schedulazione in generale può implementare

- un algoritmo di schedulazione;

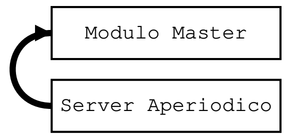


Figura 4.1: Server aperiodico che si appoggia ad un modulo master.

- un server aperiodico.

Un modulo di schedulazione in generale deve contenere al suo interno tutte le strutture dati necessarie per il suo funzionamento; a meno di limitazioni specifiche dovute alla tipologia di algoritmo implementato, deve essere possibile registrare più istanze dello stesso modulo al momento della inizializzazione del sistema.

L'implementazione di un Modulo di Schedulazione deve prescindere (nei limiti del possibile) dalla presenza di algoritmi di accesso a risorse. In effetti, durante lo sviluppo di un Modulo di Schedulazione occorre concentrarsi solo sull'aspetto dello scheduling vero e proprio. Se l'algoritmo di gestione delle risorse richiede la conoscenza delle strutture dati interne di un Modulo di Schedulazione per essere implementato in modo efficiente, allora occorrerà fare in modo che i due Moduli "si conoscano". Questa conoscenza è in ogni caso un compromesso tra modularità ed efficienza.

Mentre l'implementazione di un algoritmo di schedulazione deve prescindere dalla presenza nel sistema di altri moduli di schedulazione, l'implementazione di un server aperiodico viene tipicamente realizzata appoggiandosi ad un altro modulo di schedulazione (chiamato modulo master; vedi Figura 4.1).

Il fatto che un server aperiodico si appoggi ad un altro modulo è una diretta conseguenza dell'approccio seguito in letteratura per la implementazione dei server. In effetti, molti server aperiodici *inseriscono direttamente* i propri task all'interno delle code gestite dall'algoritmo di scheduling principale. L'approccio modulare seguito impone comunque che tale accesso sia realizzato tramite apposite funzioni di interfaccia, evitando in questo modo l'accesso

| | |
|---|---|
| Liv. 0 | Modulo XXX |
| Liv. 1 | Modulo YYY |
| Liv. 2 | Modulo WWW |
| Liv. 3 | Modulo ZZZ |

Figura 4.2: Organizzazione dei livelli del Kernel

diretto alle strutture dati interne dei moduli, che rimangono in questo modo "veramente" private. Questo modo di operare consente di disaccoppiare l'implementazione dei server aperiodici da quella dei livelli di schedulazione.

### 4.2.2 Organizzazione dei moduli

I Moduli di Schedulazione sono organizzati a *livelli*, un modulo per ogni livello. I Livelli possono essere pensati come livelli di schedulazione prioritari (l'indice 0 ha la massima priorità).

Come detto in precedenza, i Moduli di Schedulazione si spartiscono i task della applicazione basandosi sui Modelli di Task forniti al momento della creazione dei task stessi. I task vengono così *posseduti* da un Modulo di Schedulazione. Il campo `task_level` del descrittore generico di task (vedi. Sezione 4.1.1) contiene l'indice del modulo che gestisce il task.

Ciascun Modulo di Schedulazione è responsabile della gestione di tutti gli eventi che riguardano i task che gli appartengono. I task di un particolare modulo sono schedulati in background rispetto ai task appartenenti ai moduli con indice di livello minore, ed in foreground rispetto rispetto ai task appartenenti ai moduli con indice di livello maggiore.

Facendo riferimento alla Figura 4.2, i task *schedulati*[2] dal Modulo di Schedulazione XXX eseguono in foreground rispetto ai task *schedulati* dagli altri Moduli, mentre i task *schedulati* dal Modulo WWW eseguono in background

[2] Notare come nel paragrafo sia evidenziata la parola *schedulati*: i task *schedulati* da un Modulo di Schedulazione in generale sono i task posseduti dal modulo più quelli che altri Moduli vi hanno inserito (vedi la Sezione 4.2.4).

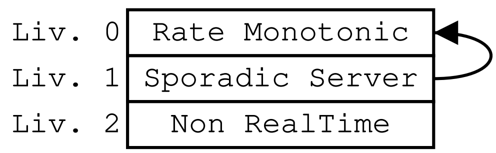


Figura 4.3: Configurazione dei Moduli secondo un approccio a priorità *statiche.*

rispetto ai task *schedulati* dai Moduli XXX ed YYY, mentre eseguono in foreground rispetto ai task *schedulati* dal Modulo ZZZ.

Ogni modulo possiede inoltre delle funzioni di interfaccia per implementare uno schedulatore ed una garanzia on-line limitatamente ai task che gli appartengono.

### 4.2.3 Potenzialita'

L'approccio seguito nella organizzazione dei moduli è molto versatile e permette di realizzare molte configurazioni anche diametralmente opposte l'una dall'altra. Di seguito sono riportati alcuni esempi di possibili configurazioni:

**Fixed Priority Classica** Questa prima configurazione (descritta in Figura 4.3) ripropone un classico approccio alla schedulazione real-time. In particolare, a priorità più alta vengono schedulati i task hard (periodici e sporadici), assieme ad i task aperiodici gestiti tramite uno Sporadic Server. In background possono infine essere presenti dei task non realtime gestiti, per esempio, da un algoritmo tipo Round Robin.

**Dual Priority Scheme** Questo secondo esempio (descritto in Figura 4.4 ) propone la possibilità di combinare assieme moduli diversi per formare configurazioni che probabilmente non erano state previste al momento della implementazione dei moduli di schedulazione. Ad esempio, è possibile definire in foreground un algoritmo di scheduling statico quale RM (con annesso un Deferrable Server); in background un algoritmo di

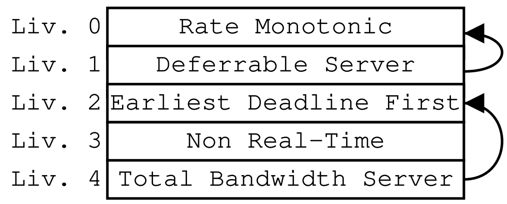


Figura 4.4: Configurazione dei Moduli secondo un approccio Dual Priority. Notare come il livello TBS possa essere posizionato come ultimo livello, in quanto l'algoritmo utilizzato intrinsecamente non necessita l'utilizzo del tempo in background lasciato libero dai moduli superiori.

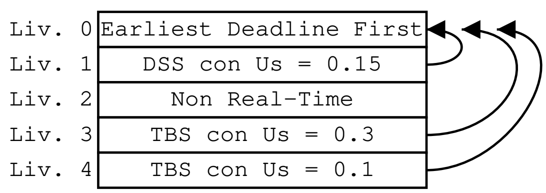


Figura 4.5: Configurazione dei Moduli secondo un approccio a priorità *dinamiche*. Per quanto riguarda i livelli TBS valgono le stesse considerazioni fatte in Figura 4.4.

scheduling dinamico quale EDF (anch'esso con un server aperiodico per gestire task "meno" critici). Una configurazione di questo tipo può ad esempio essere utilizzata per diminuire il jitter dei task più "importanti" [DW95].

**Dynamic Multiserver** Il terzo esempio (descritto in Figura 4.5) propone l'utilizzo della modularità come possibilità per creare scenari anche complessi. Nellésempio i task Hard (periodici e sporadici) vengono schedulati assieme ad una serie di task aperiodici gestiti ciascuno da un diverso server. Si noti come il modulo che implementa EDF deve essere scritto in modo da accettare task da un altro generico modulo di schedulazione, prescindendo dall'algoritmo che tale secondo modulo implementa realmente. Si noti inoltre come nel sistema siano presenti più istanze di un singolo modulo. In generale ogni modulo conoscerà so-

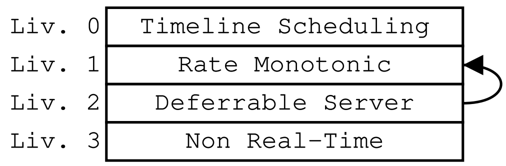


Figura 4.6: Configurazione dei Moduli secondo un approccio statico.

lo i task che gli appartengono, indipendentemente dalla configurazione reale del sistema.

**Timeline Scheduling** In questo ultimo esempio (descritto in Figura 4.6) è presentato un approccio ibrido alla schedulazione, in cui uno schedulatore statico di tipo timeline è integrato con un algoritmo a priorità fissa basato su RM. È utile far notare come l'approccio modulare consenta l'implementazione di algoritmi di scheduling diversi da quelli "classici" con accodamento prioritario. In effetti il Kernel Generico non impone alcuna struttura dati interna a chi sviluppa i Moduli di Schedulazione.

### 4.2.4 Interfaccia dei moduli

In questo paragrafo entreremo nel dettaglio architetturale della interfaccia che deve essere esportata da un Modulo di Schedulazione.

Un Modulo di Schedulazione in generale deve possedere una interfaccia che permetta di implementare un comportamento specifico per ogni evento nel sistema che possa riguardare un particolare task.

Ovviamente gli eventi che possono interessare un task in particolare sono molteplici e possono essere raggruppati in tre classi fondamentali, come descritto in Tabella 4.3:

- Level Calls - riguardano l'algoritmo implementato senza poter essere ricondotte ad un task specifico;

- Task Calls - riguardano un task specifico;

| Tipo funzione | Esempio |
|---|---|
| Level Call | garanzia, scheduler, statistiche |
| Task Call | creazione, dispatch, epilogo, fine task |
| Guest Call | creazione, dispatch, epilogo, fine guest task |

Tabella 4.3: Esempi di funzioni tipo

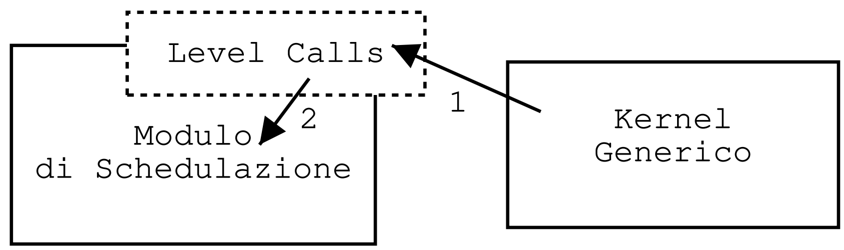


Figura 4.7: Level Calls. I quadrati a tratteggio indicano funzioni di interfaccia. I numeri vicini alle frecce schematizzano il flusso della esecuzione delle funzioni invocate tra i moduli coinvolti.

- Guest Calls - riguardano un task specifico inserito in un modulo master.

### Operazioni di livello (Level Calls)

Le operazioni globali sono funzioni del Modulo di Schedulazione che riguardano l'algoritmo implementato ma *non sono direttamente ricollegabili ad un task specifico.*

Tali funzioni di interfaccia vengono **richiamate dal Kernel Generico** per implementarne funzioni interne (cfr. Figura 4.7) non direttamente correlate con il comportamento delle primitive di sistema.

Ad esempio (vedi Sezione 8.3.1), esistono delle Level Calls per implementare:

- lo schedulatore di livello;
- la garanzia on-line;

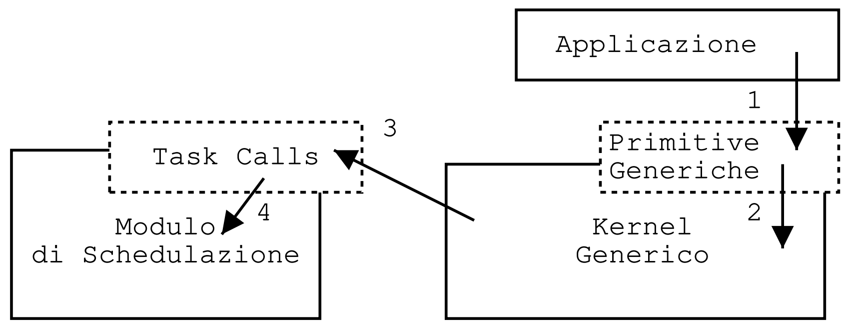


Figura 4.8: Task Calls. I quadrati a tratteggio indicano funzioni di interfaccia. I numeri vicini alle frecce schematizzano il flusso dell'esecuzione delle funzioni invocate tra i moduli coinvolti.

- le statistiche del modulo;
- le funzioni che indicano se un Modulo possa o meno accettare un task con un determinato Modello di Task.

### Operazioni sul task (Task Calls)

Le operazioni locali sono funzioni del Modulo di Schedulazione che *riguardano l' implementazione di un evento relativo ad un task specifico.* Esse vengono **richiamate dal Kernel Generico**[3] in seguito al verificarsi di eventi o in seguito alla chiamata di primitive da parte dei task utente (cfr. Figura 4.8) .

Queste funzioni sono le funzioni che condizionano il comportamento di un task e pertanto hanno il compito di **implementare il comportamento di alcune primitive** di sistema.

Ad esempio (vedi Sezione 8.3.2), esistono delle Task Calls per implementare:

- la creazione di un task;

[3]Il Kernel generico per richiamare le Task Calls ha bisogno di conoscere il livello che possiede il task su cui viene chiamata la funzione. Tale informazione é, come detto precedentemente, mantenuta nel campo `task_level` del descrittore di task.

- la terminazione di un task;
- il dispatch di un task;
- il bloccaggio per sincronizzazione;
- la sospensione per fine periodo.

### Operazioni Remote (Guest Calls)

Le operazioni remote sono funzioni del Modulo di Schedulazione che *riguardano l' implementazione di un evento relativo ad un task, di proprietà di un altro modulo, attualmente inserito all'interno del modulo corrente.* Esse vengono **richiamate da un altro Modulo di Schedulazione**[4] in seguito alla esecuzione di una funzione di interfaccia del Modulo possessore del task (cfr. Figura 4.9) .

Le operazioni remote vengono *utilizzate* tipicamente dai Moduli di Schedulazione che implementano *Server Aperiodici.* Esse sono *definite* nell'interfaccia del *modulo master* e vengono richiamate per i task che i server aperiodici inseriscono in tale modulo.

Le operazioni remote svolgono gli stessi compiti delle operazioni locali (l'unica differenza è *chi* richiama tali funzioni), pertanto esse replicano, con la stessa semantica ma con nome ovviamente modificato, le stesse funzioni presenti nelle operazioni locali.

è compito dei Server Aperiodici richiamare le operazioni remote nell'ordine corretto. L'ordine corretto di chiamata delle operazioni remote è lo stesso delle corrispondenti operazioni locali. Tipicamente un Server Aperiodico si appoggerà ad un modulo master creando in esso i propri task (passando quindi anche qui degli opportuni Modelli di Task) e chiamando in esso le opportune operazioni remote.

[4] Tale Modulo di Schedulazione per richiamare le Guest Calls ha bisogno di conoscere il livello in cui esso stesso ha inserito il task. Tale informazione non é contenuta all'interno del descrittore di task ma deve essere mantenuta nelle strutture dati interne al modulo.

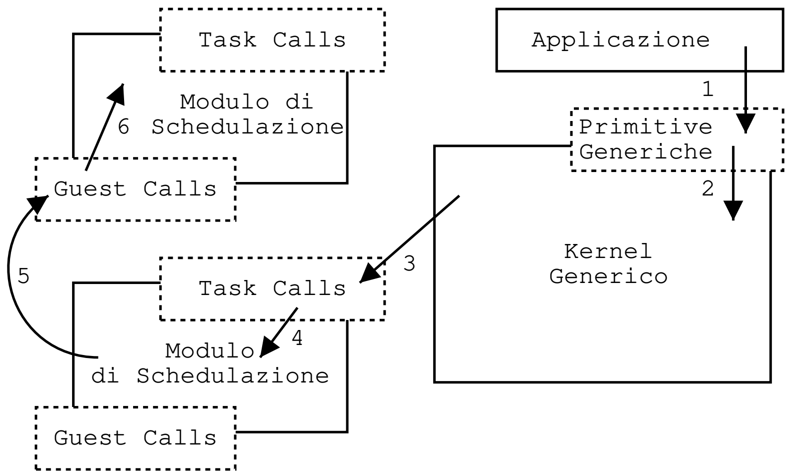


Figura 4.9: Guest Calls. I quadrati a tratteggio indicano funzioni di interfaccia. I numeri vicini alle frecce schematizzano il flusso della esecuzione delle funzioni invocate tra i moduli coinvolti.

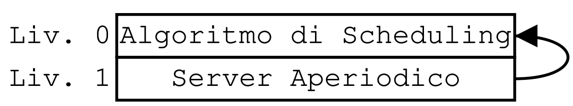


Figura 4.10: Configurazione dell'esempio di Sezione 4.2.5.

### 4.2.5 Esempio

In questo paragrafo viene mostrato un esempio utile per una migliore comprensione dell'utilizzo delle varie classi di funzioni presenti nell'interfaccia di un modulo.

L'interfaccia dei Moduli di Schedulazione verrà poi dettagliata completamente nel Capitolo 8.

L'esempio in Figura 4.10 considera un sistema in cui la configurazione dei moduli e dei livelli è formata da due moduli di schedulazione registrati in altrettanti livelli:

- al Livello 0 consideriamo un Modulo che implementa un generico Algoritmo di Scheduling;

- al Livello 1 consideriamo Modulo che implementa un generico Server Aperiodico (che inserisce i propri task all'interno Modulo registrato al livello 0).

Consideriamo inoltre 2 task all'interno del sistema:

- Task A e' un task creato all'interno del Modulo che implementa l'Algoritmo di Scheduling (registrato a livello 0) e pertanto `task_level` = 0.

- Task B e' un task creato all'interno del Modulo che implementa il Server Aperiodico (registrato a livello 1), e pertanto `task_level` = 1. Inoltre, il task B èinserito nel livello 0 tramite le funzioni Guest Oriented.

Entrambe i task vengono schedulati dallo scheduler del livello 0 (il Task A perchè in esso è stato creato, il Task B perchè, sebbene creato nel livello 1, è stato inserito nel livello 0 tramite le funzioni Guest Oriented).

Al momento in cui la procedura di scheduling sarà invocata, lo scheduler del Kernel generico richiamera' in sequenza i level_scheduler di ogni livello, partendo dallo scheduler di indice 0; in particolare, lo scheduler del livello 0 sceglierа' quale task realmente schedulare tra i due che puo' gestire (A e B).

Se la procedura di scheduling indica il Task A come prossimo task da schedulare, il Kernel Generico chiamerà la Task Call dispatch del Modulo registrato al livello 0 (in quanto il campo `task_level` del Task A vale 0).

Se invece la procedura di scheduling indica il Task B, il Kernel Generico chiamerà la task Call dispatch del Modulo registrato al livello 1 (in quanto il campo `task_level` del Task B vale 1), che a sua volta, per gestire l'evento dispatch del proprio task, tra le altre cose dovrà richiamare la Guest Call dispatch del livello 0.

### 4.2.6 Commento

Esistono moltissimi modi per implementare all'interno di un kernel un server aperiodico. Ciascun metodo ha vantaggi e svantaggi. In questo paragrafo verrà data una giustificazione dell'approccio utilizzato in questa particolare implementazione.

Occorre innanzitutto ricordare che uno degli obiettivi fondamentali seguiti nello sviluppo del Kernel è quello della modularità, intesa come possibilità di poter cambiare e modificare i Moduli di Schedulazione in modo indipendente dal Kernel Generico, dagli altri moduli e dalle applicazioni. Questo obiettivo è stato raggiunto disaccoppiando i Task dall'algoritmo che ne implementa la schedulazione tramite i Modelli di Task. Inoltre l'algoritmo e la gestione del task all'interno del Kernel Generico è totalmente modularizzata. In questo modo chi scrive dei Moduli è costretto solamente a dover rispettare l'interfaccia proposta dal Kernel Generico, senza avere limitazioni nè sulle strutture dati utilizzabili internamente, nè sulle tecniche di scheduling da adottare.

In effetti gli algoritmi di schedulazione ed i server aperiodici utilizzano la *stessa* interfaccia, e questo in definitiva rende possibile la realizzazione, in piena libertà, di moduli anche leggermente diversi da quelli standard, quali, ad esempio, algoritmi di schedulazione che si appoggiano ad altri moduli tramite le guest calls, oppure moduli che implementano in modo integrato sia server aperiodici che algoritmi di schedulazione.

Esistono comunque altri possibili approcci. Durante il disegno della architettura sono stati presi in considerazione altri approcci seguiti in letteratura (vedi Capitolo 2) che, sebbene modulari, non offrivano tutte le possibilità fornite dall'approccio seguito.

## 4.3 Moduli di Gestione delle risorse

### 4.3.1 Obiettivi dei Moduli di Gestione delle Risorse

I Moduli di Gestione delle Risosrse sono solitamente utilizzati per implementare moduli del sistema che non si occupano di gestire direttamente la schedulazione dei task ma che necessitano comunque di informazioni che devono essere fornite alla creazione ed alla distruzione dei singoli task.

Moduli di questo tipo sono ad esempio quelli che implementano protocolli di accesso a risorse condivise (che a volte richiedono la specifica di alcuni parametri, quale il ceiling del sistema, oppure l'insieme dei mutex utilizzati da un certo task, ecc...), oppure i Moduli che implementano un file system che supportano una specifica della banda su disco da garantire ad ogni task.

### 4.3.2 Organizzazione dei moduli e loro interfaccia

In modo analogo a quanto detto per i Moduli di Schedulazione, l'organizzazione scelta per i Moduli di Gestione delle Risorse è ancora a livelli, un modulo per ogni livello.

In questo, però i livelli non hanno un ordinamento prioritario come succedeva per i Moduli di Schedulazione. Il numero di livello di un Modulo di Gestione delle Risorse influenza solo l'ordine con cui a tali moduli vengono notificati gli eventi di interesse.

Per quanto riguarda l'interfaccia esportata nei confronti del Kernel generico si può dire che gli unici eventi di interesse rilevati da un Modulo di Gestione delle Risorse sono gli eventi di creazione e di fine task.

In modo ancora analogo ai Moduli di Schedulazione, al momento della creazione di un task possono essere specificati uno o più Modelli di Risorsa che verranno forniti in modo adeguato ai Moduli competenti.

### 4.3.3 Commento

L'interfaccia fornita da un Modulo di Gestione delle Risorse non è ovviamente completa, in quanto, oltre alle funzioni di interfaccia con il Kernel Generico, il Modulo dovrà fornire una serie di costanti, tipi e funzioni necessarie per interagire con la risorsa da gestire.

L'approccio seguito nel caso in cui a sua volta la risorsa da gestire o il protocollo da implementare debbano essere resi modulari (come ad esempio succede nella implementazione dei protocolli di accesso a risorse condivise) è il seguente.

I Moduli che devono supportare una interfaccia modulare estendono l'interfaccia generica di un Modulo di Gestione delle Risorse aggiungendo la parte di interfaccia mancante (utilizzando un linguaggio simile a quello utilizzato nella programmazione a oggetti, si può dire che tali moduli ereditano l'interfaccia esportata da una classe base del Kernel Generico estendendola per ottenere una ulteriore classe dalla quale derivare le effettive implementazioni).

Gli esempi di Moduli di Gestione delle Risorse realizzati attualmente nel Kernel riguardano l'implementazione dei Protocolli di Accesso a Risorse Condivise (cfr. Sezione 4.4).

## 4.4 Moduli che implementano protocolli di accesso a risorse condivise

### 4.4.1 Obiettivi

I Protocolli di Accesso a Risorse Condivise sono implementati tramite una estensione dei Moduli di Gestione delle Risorse ed hanno i seguenti obiettivi fondamentali:

- devono essere il più possibile modulari, nel senso che devono prescindere il più possibile dalla implementazione dei Moduli di Schedulazione che possiedono e gestiscono i task che usano un dato protocollo. Questo implica anche che i Moduli di Schedulazione devono essere a loro volta sviluppati in modo indipendente dai protocolli di accesso.

- devono poter coesistere contemporaneamente, nel senso che si deve rendere possibile a due insiemi disgiunti di task di utilizzare due distinti protocolli in modo indipendente.

- devono poter esportare una interfaccia compatibile con i mutex definiti nello standard POSIX.

### 4.4.2 Mutex, Parametri e Modelli

Per seguire l'obiettivo della compatibilità con lo standard POSIX, l'approccio architetturale seguito nella implementazione dei protocolli è stato quello di predisporre una infrastruttura compatibile a livello funzionale con quella dello standard.

Seguire un simile approccio significa che:

- occorre definire un tipo di dato `mutex_t` (i nomi sono diversi dallo standard ma sono ad esso direttamente riconducibili) contenente tutte le strutture dati interne necessarie per implementare un mutex con un generico protocollo[5];

- il protocollo effettivamente utilizzato da un mutex viene stabilito al momento della chiamata di una funzione di inizializzazione;

- i parametri riguardanti il protocollo da utilizzare devono essere passati in appositi Modelli di Risorsa, per quanto riguarda i parametri da

[5] Ovviamente ogni protocollo avrà dei parametri interni che verranno memorizzati o nelle strutture dati interne del modulo o in una zona di memoria allocata dinamicamente il cui puntatore si troverà nella struttura `mutex_t`.

fornire alla creazione di un task[6], ed in appositi Modelli di Mutex, per quanto riguarda i parametri da fornire alla inizializzazione dei mutex[7];

- per ottenere un mutex con queste caratteristiche (cioè fondamentalmente quella di avere un protocollo "polimorfico") è necessario disporre di una interfaccia comune dalle quali derivare le verie implementazioni (e questo, è bene ricordare, è stato ottenuto tramite la derivazione dell'interfaccia dei mutex dai Moduli di Gestione delle Risorse.

### 4.4.3 Problema della inversione di priorità

Uno dei problemi più grossi in cui ci si imbatte nell'implementare un Protocollo di Accesso a Risorse Condivise è il problema dell'Inversione di Priorità[8] [SRjPL90].

Tale inconveniente è stato brillantemente risolto utilizzando intelligenti stratagemmi che influenzano la schedulazione dei task. Quello che tipicamente accade è che in alcuni protocolli il task indicato dallo scheduling[9] non è sempre quello su cui viene fatto il dispatching[10].

Protocolli di questo tipo sono ad esempio i protocolli

- Priority Inheritance

[6] Lo standard POSIX specifica la possibilità di fornire, (all'interno del tipo `pthread_attr`) un parametro che indica la priorità del task, che verrà utilizzata, oltre che per lo scheduling, come priorità per i mutex che utilizzano il protocollo Priority Ceiling (o Prioriy Protect nel gergo POSIX).

[7] Lo standard POSIX specifica il protocollo di un mutex all'interno del tipo `pthread_mutexattr_t`, predisponendo per il protocollo Priority Ceiling l'indicazione della massima priorità dei thread che utilizzano tale mutex.

[8] Tale fenomeno si verifica quando un processo ad alta priorità viene bloccato da un processo a più bassa priorità per un intervallo di tempo indefinito anche a causa dell'intervento di altri processi a priorità intermedia che possono effettuare preemption sul processo a bassa priorità.

[9] con scheduling, si ricorda, si intende l'insieme di regole che determinano la scelta del processo da mandare in esecuzione tra i processi pronti

[10] con dispatching, si ricorda, si intende l'assegnazione effettiva del processore al processo pronto schedulato.

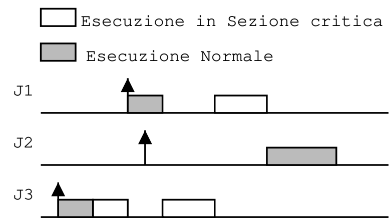


Figura 4.11: Esempio di schedulazione utilizzando il protocollo di Priority Inheritance: il task $J_3$ arriva entrando in sezione critica; $J_1$ fa preemption; $J_2$ arriva ma rimane in coda pronti essendo $J_1$ a priorità maggiore; al momento in cui $J_1$ tenta di entrare in sezione critica, $J_3$ ne eredita la priorità fino all'uscita della stessa.

- Priority Ceiling

in cui l'ordinamento dei task nella coda pronti viene modificato per far si che il task bloccante a bassa priorità erediti *momentaneamente* la priorità del task bloccato ad alta priorità (cfr. Figura 4.11) [SRjPL90].

I problemi che si incontrano in una implementazione modulare di tali protocolli sono i seguenti:

- il Kernel Generico non possiede concetti quali priorità, periodi, deadline, normalmente utilizzati come parametro discriminante per l'accodamento dei task;

- il Kernel Generico, e tantomeno i Moduli che implementano i protocolli, non conoscono nè il metodo di accodamento utilizzato per gestire le code dei task, nè la posizione in memoria di tali code[11] per eventualmente accedervi;

- uno degli obiettivi dello sviluppo dei protocolli di accesso a risorse condivise è l'indipendenza della implementazione dei protocolli dal resto dei Moduli, quindi, seppur possibile, è sconsigliabile agire direttamente sulle strutture dati interne di un altro Modulo.

[11] sempre che tali code esistano!!!

### 4.4.4 Soluzioni proposte

Le soluzioni possibili per evitare l'inversione di priorità in sistemi modulari sono fondamentalmente due. La prima non necessita di un supporto specifico da parte del Kernel Generico, e pertanto può essere sempre implementata. La seconda necessita invece di alcuni nuovi concetti, che hanno poi trovato reale implementazione all'interno del kernel.

#### Prima Soluzione

La prima soluzione è quella classicamente adottata nelle implementazioni non modulari dei kernel real-time.

In pratica si implementa l'algoritmo di schedulazione ed il protocollo di accesso a risorse condivise in modo tale che le primitive di mutua esclusione (ad esempio il lock e l'unlock dei mutex) possano accedere direttamente alle strutture dati interne del Modulo di Schedulazione, quali le code dei processi ed il campo in cui viene memorizzata la priorità di un task.

Se si adotta questo approccio in un'ottica modulare si ottiene come risultato che i Moduli di Schedulazione ed i Moduli che implementano Protocolli di Accesso a Risorse Condivise devono essere realizzati in modo tale che almeno uno dei due moduli conosca le strutture dati interne dell'altro[12].

Questo approccio ha dei vantaggi, in quanto:

- l'implementazione dei Moduli può essere realizzata in modo efficiente grazie alla conoscenza precisa delle strutture dati interne;
- la garanzia del sistema potrà tenere conto di eventuali limitazioni o casi particolari dovuti ai tempi di bloccaggio dei task.

Lo svantaggio di tale soluzione riguarda l'utilizzo finale di tali Moduli, in quanto essi potranno essere utilizzati solo "assieme", come se fossero una cosa

[12] Tipicamente sara' il Modulo che implementa il protocollo a conoscere le strutture dati del Modulo di Schedulazione.

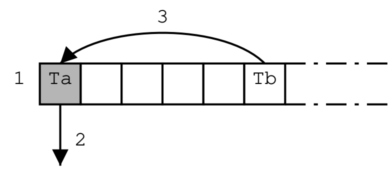


Figura 4.12: Implementazione della ereditarietà mediante inserimento fuori ordine: (1) Ta si blocca nell'accesso ad una risorsa; (2) Ta va in coda bloccati; (3) Tb sostituisce momentaneamente il task ad alta priorità.

sola, eliminando la possibilità di combinare moduli non scritti esplicitamente per lavorare assieme.

### Seconda Soluzione

L'unico metodo per evitare che due Moduli siano costretti a condividere le proprie strutture dati interne è quello di far si che i meccanismi di ereditarietà delle priorità vengano supportati direttamente dal Kernel Generico in modo *indipendente* dall'implementazione dei Moduli.

Un'idea simile è stata utilizzata nello sviluppo di altri progetti, come ad esempio in CPU Inheritance (vedi [FS96]).

L'approccio seguito nello sviluppo del Kernel Generico è leggermente diverso dall'approccio citato sopra. L'idea di base nasce dalla considerazione che nelle implementazioni tradizionali lo scambio delle priorità viene spesso realizzato inserendo i task *fuori ordine* in code prioritarie[13].

Facendo riferimento alla Figura 4.12, il task Tb a bassa priorità si sposta nella posizione ad alta priorità occupata dal task bloccato Ta.

L'idea alla base del meccanismo realizzato nel nuovo Kernel prevede che il task ad alta priorità bloccato rimanga in coda pronti e indichi un altro task (ombra, o *shadow*) da eseguire in sua sostituzione al momento della dispatch (vedi Figura 4.13).

[13]Con il termine *inserimento fuori ordine* si intende un inserimento in coda senza rispettare il normale ordinamento basato sulle priorità contenute nei descrittori di task.

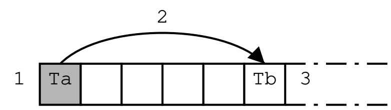


Figura 4.13: Implementazione della ereditarietà mediante utilizzo del meccanismo degli shadows: (1) Ta si blocca nell'accesso ad una risorsa; (2) Ta indica il task sostitutivo (ombra) rimanendo in coda pronti; (3) Tb non si accorge di nulla.

In questo modo il Task bloccante non si accorge di ereditare la priorità del task bloccato, disaccoppiando l'implementazione degli algoritmi di scheduling da quella dei protocolli di accesso a risorse condivise.

### 4.4.5 Cenni sulla implementazione della seconda soluzione

In questo paragrafo è presentata una breve descrizione di una possibile implementazione della seconda soluzione proposta.

Per implementare il meccanismo degli shadow occorre innanzitutto distinguere il task indicato dallo scheduling dal task su cui verrà effettivamente effettuato il dispatching.

Supponiamo che il Kernel Generico disponga di un indicatore del task in esecuzione. Tale indicatore deve essere sdoppiato in due: chiameremo `exec` ed `exec_shadow` i due nuovi indicatori. Il significato dei due indicatori è il seguente:

- `exec` indica il task scelto dalla procedura di scheduling;
- `exec_shadow` indica il task su cui effettivamente viene effettuato il dispatch.

Viene inoltre aggiunto al descrittore di task un nuovo campo, chiamato `shadow`: la variabile shadow indica il task ombra da eseguire in sostituzione di quello schedulato (vedi Figura 4.14).

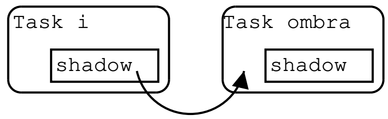


Figura 4.14: Significato del campo `shadow` del dsecrittore generico di task.

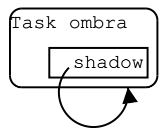


Figura 4.15: Tipico scenario in cui il campo `shadow` di un task indica il task stesso (non c'è sostituzione).

Normalmente il campo shadow di un task indica il task stesso, ovvero non segnala alcuna sostituzione (vedi Figura 4.15).

Quando un task Ta si blocca, ad esempio su una risorsa mutuamente esclusiva occupata da un task Tb il campo `shadow` del task Ta viene settato al valore Tb (vedi Figura 4.16). In generale, più task possono bloccarsi sulle stesse risorse, formando così dei grafi di attesa (cfr. Figura 4.17).

Per quanto riguarda la schedulazione, i Moduli di Schedulazione continuano ad indicare il task da eseguire *prescindendo* dai valori contenuti nel campo `shadow` dei descrittori di task. Il Kernel Generico pertanto interroga i Moduli di Schedulazione alla ricerca di un task da eseguire e pone l'indicazione del task scelto all'interno della variabile `exec`. Subito dopo aver effettuato la schedulazione, il Kernel Generico scorre il grafo degli shadow finchè non trova un task per cui il campo `shadow` indichi il task stesso. A questo punto

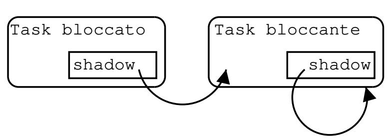


Figura 4.16: Tipico scenario in cui un task bloccato rimane in attesa di una risorsa facendo ereditare la propria priorità al task bloccante.

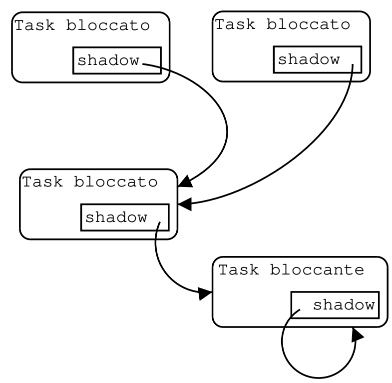


Figura 4.17: Utilizzando il meccanismo degli shadows si possono formare dei grafi di attesa (nell'ordine, il task bloccante acquisisce una risorsa, richiesta poi dal primo task bloccato, il quale a sua volta possiede un'altra risorsa su cui in un tempo successivo si bloccano gli altri due task).

il Kernel generico setta la variabile `exec_shadow` per farla puntare al task trovato scorrendo il grafo ed infine effettua la dispatch di tale task.

### 4.4.6 Deadlock

Uno dei vantaggi dell'implementazione proposta è la possibilità di identificare situazioni di stallo (deadlock). Una situazione di deadlock è infatti caratterizzata dalla presenza di un ciclo all'interno del grafo degli shadow.

Per risolvere il problema del deadlock si possono adottare sia protocolli che garantiscono l'assenza di deadlock, quali, ad esempio, Priority Ceiling [SRjPL90] ed SRP [Bak90], sia protocolli che non impediscono il verificarsi del deadlock, ma che lo prevengono impedendo al task che crea il deadlock di entrare in sezione critica[14].

In ogni caso, l'utilizzo della variabile shadow permette di implementare in modo semplice il riconoscimento della formazione dei cicli all'interno del grafo delle attese. In pratica è sufficiente controllare la condizione di deadlock

[14]Questo secondo approccio é quello seguito da POSIX nella definizione dell'interfaccia dei mutex.

al momento del settaggio di una variabile shadow. Tale controllo non puó essere svolto dal Kernel Generico, ma è lasciato a carico dei Moduli che implementano i protocolli di accesso alle risorse condivise.

### 4.4.7 Overhead

Può essere utile a questo punto fare alcune considerazioni riguardanti l'overhead che l'approccio mediante shadow introduce all'interno del Kernel Generico.

In generale (stime più accurate sono trattate nel Capitolo 13) l'overhead nella dispatch e' presumibilmente basso, in quanto occorre scorrere una lista concatenata di indici interi, tipicamente costituita da 1-2 elementi.

L'overhead più significativo è quello che si ha al momento del bloccaggio, in quanto l'implementazione di un generico protocollo potrebbe richiedere la modifica del campo `shadow` in più di un task.

# Chapter 5

# La OS Lib e la sua interfaccia

In questo capitolo verranno richiamati una parte dei concetti e delle funzioni di interfaccia della OS Lib su cui si è appoggiato lo sviluppo del Kernel Generico. La parte dell'interfaccia presentata è quella strettamente necessaria per comprendere l'architettura degli strati superiori.

Per ulteriori informazioni riguardo alla OS Lib si può fare riferimento al testo (vedi [Abe00]).

## 5.1 Il Tempo

Uno dei compiti primari svolti dalla OS Lib è quello di fornire un meccanismo per la gestione del tempo all'interno del sistema.

In molti approcci classici il tempo di CPU è suddiviso in piccole fette che rappresentano la risoluzione temporale del sistema; tutte le grandezze temporali all'interno del sistema vengono espresse in multipli di tale unità temporale (tick).

All'interno della OS Lib invece il tempo viene gestito con la risoluzione massima ottenibile dal sistema ospite[1].

[1] La lettura del tempo avviene con la precisione del realtime clock del PC (pari a circa 840 ns). La effettiva gestione degli eventi temporali generati per un determinato istante

```
struct timespec {
    long tv_sec; /* Seconds */
    long tv_nsec; /* Nanoseconds */
};
```

Figure 5.1: Definizione della struct timespec.

Per manipolare informazioni temporali sono state predisposte due strutture dati:

`TIME` è un intero a 32 bit che rappresenta una informazione temporale espressa in microsecondi ($\mu s$).

`struct timespec` è una struttura che rappresenta un tempo espresso in secondi e nanosecondi. La struttura è compatibile con lo standard POSIX ed è dichiarata come in Figura 5.1.

La OS Lib rende inoltre disponibile una funzione da utilizzare per *leggere* il tempo di sistema. Tale funzione deve essere richiamata ad interruzioni disabilitate[2] e possiede il seguente prototipo:

```
TIME ll_gettime(int mode, struct timespec *tsres);
```

Il valore ritornato rappresenta il tempo trascorso dalla partenza del sistema[3], espresso in microsecondi[4]. Il primo parametro della funzione può avere vari valori, in questo contesto l'unico interessante è la costante `TIME_EXACT`.

avviene invece con un certo ritardo (vedi Sezione 5.7).

[2] Le applicazioni che utilizzano il Kernel generico possono utilizzare la primitiva `sys_gettime` che semplicemente richiama la funzione ad interruzioni abilitate.

[3] Ovvero dalla chiamata della funzione `ll_init`.

[4] Notare come un tempo espresso in microsecondi ha un lifetime di circa 1 ora e 11 minuti; dopo quel tempo il valore ritornato é privo di significato (o meglio, il valore si azzera raggiunto il limite massimo).

Il secondo parametro rappresenta il tempo trascorso dalla partenza del sistema espresso con una variabile di tipo `timespec`. Se il secondo parametro vale `NULL` viene restituito solo il tempo espresso in microsecondi.

La lettura del tempo, dovendo accedere al bus di sistema, è abbastanza lenta ed occupa la CPU per un tempo dell'ordine di grandezza di circa 2.8 microsecondi (vedi Capitolo 13).

La gestione del tempo funziona correttamente solo a patto che le interruzioni non rimangano disabilitate per un periodo più lungo del tempo limite di programmazione del clock real-time (circa 50 ms).

Infine, nel file `include/ll/sys/ll/time.h` la OS Lib mette a disposizione una serie di macro utili per manipolare i tempi espressi con la struttura timespec, nonchè per implementare conversioni tra tale struttura ed il tipo `TIME`.

## 5.2 Gli Eventi

L'interfaccia esportata dalla OS Lib verso il Kernel Generico è una interfaccia ad eventi. Con interfaccia ad eventi si intende il fatto che il sistema può richiedere l'esecuzione (ad interruzioni disabilitate) di una funzione ad un determinato istante temporale (*tempo di attivazione* dell'evento). La funzione è una classica funzione in linguaggio C, di cui è possibile specificare un parametro da passare al momento della chiamata. Anche gli interrupt vengono visti con una interfaccia simile agli eventi (cfr. Sezione 5.3).

La OS Lib gestisce internamente una coda di eventi sufficientemente ampia da contenere tutti gli eventi pendenti generati all'interno del sistema. Gli strati superiori in generale non hanno modo di sapere come tale coda è implementata. In ogni caso, è bene sapere che a causa di limitazioni interne, la OS Lib può rifiutarsi di accettare nuovi eventi temporali.

L'ordine di esecuzione di eventi aventi lo stesso tempo di attivazione non è specificato.

```
void event_setprologue(void (*p)(void));
void event_setepilogue(void (*p)(void));
int event_post(struct timespec time,
               void (*handler)(void *p),
               void *par);
int (*event_delete)(int index);
```

Figure 5.2: Funzioni di gestione degli eventi.

La OS Lib garantisce che un evento non scatterà mai prima del suo tempo di attivazione. Quello che invece la OS Lib non può garantire è il momento esatto in cui tali eventi verranno eseguiti. In generale sarà **sempre presente un ritardo** dell'ordine di qualche microsecondo tra il tempo di attivazione ed il tempo in cui l'evento viene realmente eseguito.

Di seguito verranno descritte le funzioni (da chiamare ad interruzioni disabilitate) messe a disposizione dalla OS Lib per gestire gli eventi (le loro dichiarazioni sono presentate in Figura 5.2) :

**event_setprologue** Permette di indicare una funzione da richiamare quando la OS Lib assume il controllo della CPU per gestire un evento[5], prima della gestione dell'evento stesso. Se questa funzione non è mai richiamata, nessun prologo viene richiamato.

**event_setepilogue** Permette di indicare una funzione da richiamare quando la OS Lib ha terminato l'esecuzione degli eventi e si appresta a ritornare alla esecuzione dei task utente. Se questa funzione non è mai richiamata, nessun epilogo viene richiamato.

**event_post** Questa funzione indica al sistema una funzione `p` con relativo parametro `par` da eseguire ad uno specifico tempo `time`. L'identificatore

[5]Ad esempio, quando arriva una interruzione hardware oppure quando arriva il momento di eseguire un evento richiesto tramite una `event_post`.

dell'evento creato viene restituito come valore di ritorno della event_post[6]. Se la OS Lib non può momentaneamente gestire la richiesta, viene restituito un valore `-1`.[7]

**event_delete** Questa funzione serve per cancellare un evento all'interno della coda eventi della OS Lib. Essa riceve come parametro un identificatore di evento, e ritorna `1` se l'evento è stato cancellato con successo, `-1` se l'identificatore passato è non significativo.

## 5.3 Gestione degli interrupt

La OS Lib unifica assieme la gestione delle interruzioni con la gestione degli eventi. Per gli strati superiori una interruzione è vista come la chiamata di una funzione di evento. Il parametro passato all'evento è un intero[8] rappresentante il numero della interruzione da gestire.

Per specificare la funzione da chiamare (handler) al momento della gestione di un interrupt si utilizza la funzione `irq_bind`, il cui prototipo e' presentato di seguito:

```
int irq_bind(int irq, void (*handler)(void *p), DWORD flags);
```

I parametri di questa funzione sono, nell'ordine:

`irq` il numero di interruzione di cui si sta specificando handler (da 1 a 15[9]);

[6]L'identificatore creato è significativo dal momento in cui viene restituito dalla event_post fino a quando esso viene effettivamente viene gestito o una event_delete su quell'identificatore viene chiamata. Un identificatore può essere riutilizzato dalla OS Lib quando diviene non più significativo.

[7]Le applicazioni che utilizzano il Kernel Generico devono utilizzare la funzione kern_event_post (vedi 7.6.1).

[8]Il parametro passato ad un evento é di tipo `void *`. Per ottenere il numero della interruzione é necessario utilizzare un cast esplicito. Se `par` é il parametro della funzione evento, il numero della interruzione da gestire si ottiene con `(void *)par`.

[9]L'interrupt 0 collegato al real time clock é gestito internamente dalla OS Lib.

`handler` la funzione che viene richiamata all'arrivo di una interruzione;

`flags` specificano alcuni possibili comportamenti della funzione. In particolare sono definiti 2 flag:

- `INT_PREEMPTABLE` specifica che la funzione può essere interrotta da altri interrupt a priorità più alta.
- `INT_FORCE` se specificato impone la modifica di un handler per una interruzione già impostata (se non specificato ogni tentativo di modificare un handler già impostato fallisce).

Per *liberare* un interrupt su cui è stata precedentemente chiamata una funzione `irq_bind` occorre richiamare la funzione `irq_bind` stessa con un `handler` uguale a `NULL` ed un parametro `flags` uguale a `INT_FORCE`.

Il valore ritornato dalla irq_bind è `1` se il settaggio dell'handler è andato a buon fine, `-1` se `l'handler` era già stato specificato in precedenza ed adesso non era stata specificato il flag `INT_FORCE`.

La OS Lib mette inoltre a disposizione seguente funzione:

```
int ll_ActiveInt();
```

Tale funzione può venire utilizzata per sapere se la sua esecuzione sta avvenendo all'interno di un event handler (sia esso dovuto ad un interrupt o generato dal sistema con una `event_post`) o meno. Nel primo caso il valore ritornato è un intero diverso da 0, nel secondo caso viene ritornato 0.

## 5.4 Disabilitazione delle Interruzioni

La OS Lib mette a disposizione quattro funzioni per implementare la disabilitazione delle interruzioni:

**void cli()** Disabilita le interruzioni;

**void sti()** Riabilita le interruzioni;

**SYS_FLAGS ll_fsave()** Disabilita le interruzioni salvando lo stato dei flags dentro il valore ritornato dalla funzione;

**void ll_frestore(SYS_FLAGS f)** Ripristina la situazione dei flags precedente alla ll_fsave che ha fornito il parametro.

Le ultime due funzioni utilizzano un tipo opaco[10] SYS_FLAGS utilizzato per contenere i flags del processore; tali funzioni vengono utilizzate per disabilitare le interruzioni quando non è possibile sapere lo stato della abilitazione al momento della disabilitazione.

## 5.5 I cambi di contesto

Uno degli obiettivi della OS Lib è quello di fornire una interfaccia semplice per la gestione della multiprogrammazione, in particolare per gestire il cambio di contesto. Attualmente la OS Lib utilizza le possibilità offerte dai processori della famiglia Intel 80386 e superiori per implementare il cambio di contesto in hardware. Tali funzionalità sono esportate verso gli strati superiori tramite la interfaccia descritta nei paragrafi seguenti.

Per prima cosa la OS Lib esporta un tipo opaco chiamato `CONTEXT`. All'interno di tale tipo vengono mascherati i registri macchina nonchè tutte le informazioni necessarie per effettuare un cambio di contesto.

La OS Lib fornisce quindi una serie di funzioni per inizializzare e distruggere un oggetto `CONTEXT`. L'interfaccia di tali funzioni è la seguente:

```
CONTEXT ll_context_create(void (*body)(void *p), BYTE *stack,
void *parm, void (*killer)(void),WORD ctrl);
```

[10] Un tipo opaco è un tipo la cui struttura interna è privata e non visibile all'esterno delle funzioni che lo utilizzano.

Questa funzione crea un nuovo contesto e lo restituisce all'interno del valore di ritorno.

Si suppone che il numero di contesti che un kernel basato sulla OS Lib possa chiedere sia sempre minore dei contesti creabili; in altre parole, si suppone che la chiamata a `ll_context_create` abbia sempre successo (sarà cura degli strati superiori non creare troppi contesti...). I parametri passati alla funzione sono:

`body` questo parametro è l'indirizzo della prima istruzione da eseguire quando verrà effettuato un cambio sul nuovo contesto appena creato ed è rappresentato da un puntatore ad una funzione che riceve un puntatore a void come parametro.

`stack` questo parametro è il puntatore all'aria di memoria utilizzabile come stack da parte del nuovo contesto creato.

`parm` questo è il parametro passato all'interno dello stack alla funzione il cui indirizzo della prima istruzione da eseguire è contenuto nel parametro `body`.

`killer` questo parametro indica la funzione da richiamare al ritorno della funzione specificata nel parametro `body`. Tipicamente questa funzione si occupa di liberare il contesto e far terminare l'eventuale task ad esso associato.

`ctrl` questo parametro è presente ma attualmente non è utilizzato.

Il tipo `CONTEXT` viene tipicamente utilizzato come contenitore per mantenere lo stato dei task all'interno del sistema. Tipicamente, ad ogni task del sistema corrisponde un contesto distinto. Quando una variabile di contesto non viene più utilizzata (ad esempio è terminato il task ad essa associato) è consigliabile liberare il contesto allocato tramite una chiamata alla funzione `ll_context_delete`, il cui prototipo è illustrato di seguito:

```
void ll_context_delete(CONTEXT c);
```

Infine per effettuare realmente i cambi di contesto (ad esempio all'interno delle primitive che prevedono preemption) dovranno essere utilizzate le seguenti funzioni:

```
CONTEXT ll_context_from(void);
void ll_context_to(CONTEXT c);
```

La prima funzione dovrà essere richiamata all'inizio della procedura che cambia il contesto, subito dopo la disabilitazione delle interruzioni. Essa ritorna come valore il contesto corrente.

La seconda funzione è invece quella che effettivamente effettua il cambiamento di contesto su quello specificato come parametro. Se tale funzione è chiamata all'interno di un event handler (cioè in un punto in cui la primitiva `ll_ActiveInt()` riitorna un valore diverso da `0`) il cambio di contesto verrà posticipato a subito dopo la chiamata dell'epilogo degli eventi (vedi Sezione 5.2).

## 5.6 La LIBC

Uno dei componenti di cui è composta la OS Lib è un sottoinsieme della LIBC[11]. In particolare, vengono implementate all'interno della OS Lib l'insieme delle funzioni standard del linguaggio C che non dipendono da caratteristiche del sistema operativo, quali, ad esempio, le funzioni di manipolazione delle stringhe, le funzioni riguardanti la copia della memoria, i codici di errore, ecc.

[11] Con LIBC si intende l'insieme di funzioni più o meno standard che vengono incluse in ogni distribuzione dei compilatori C. Oltre alle funzioni specifiche dello standard C o dello standard POSIX, sono spesso presenti altre funzioni di utilità nonchè primitive non-standard dipendenti dal sistema operativo.

```
/* Strutture dati e costanti utilizzate da ll_init */
#define LL_PERIODIC 0

struct ll_initparms {
    DWORD mode;
    TIME tick;
};

/* Funzioni di inizializzazione e terminazione */
void *ll_init(struct ll_initparms *l);
void event_init(void);
void ll_end(void);
void ll_abort(int code);
```

Figure 5.3: Funzioni di inizializzazione della OS Lib.

Per realizzare l'intera LIBC, un Kernel basato sulla OS Lib dovrà implementare le restanti parti, quali ad esempio le funzioni di I/O con i files, le funzioni che gestiscono i processi ed i threads, ecc.

Per una descrizione delle funzioni realmente implementate si faccia riferimento a (vedi [Abe00]).

## 5.7 Funzioni di inizializzazione della OS Lib

Le strutture dati interne alla OS Lib che permettono di gestire eventi ed interruzioni non vengono inizializzate alla partenza del sistema (per avere una idea della sequenza di inizializzazione del sistema si faccia riferimento alla Sezione 5.9).

Per effettuare tale inizializzazione è necessario richiamare all'interno della applicazione due apposite funzioni, chiamate `ll_init` e `event_init`, la cui interfaccia è riportata in Figura 5.3.

La funzione `ll_init` deve essere sempre chiamata prima della `event_init`.

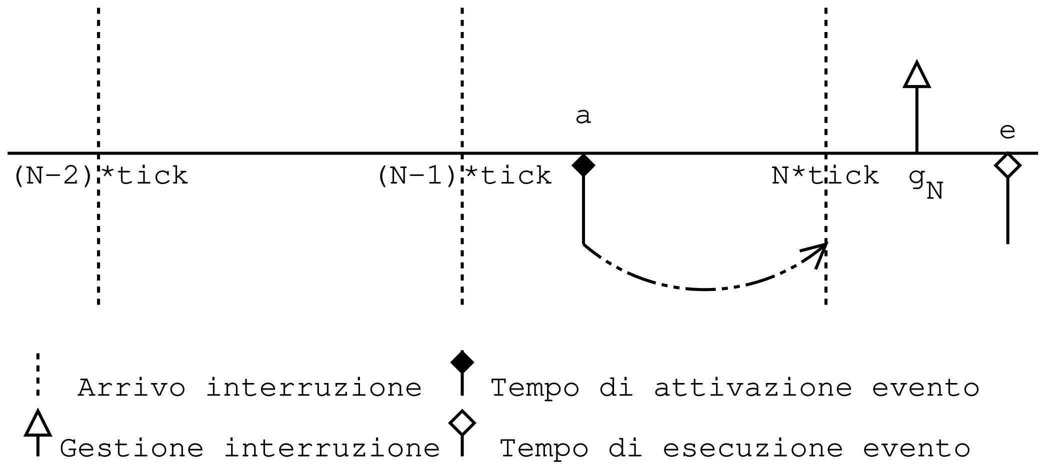


Figure 5.4: Clock real-time programmato in modo *periodico*. Gli eventi vengono effettivamente eseguiti all'arrivo del primo interrupt successivo al tempo di attivazione. Il tempo che passa tra $a$ ed $N * tick$ è dovuto alla approssimazione periodica effettuata dalla OS Lib. Il tempo che passa da $N * tick$ ad $g_N$ è un ritardo dovuto ad eventuali disabilitazioni delle interruzioni al momento dell'arrivo dell'interrupt. Il tempo che passa tra $g_N$ ed $e$ è dovuto in generale alla esecuzione di altri eventi durante la stessa interruzione.

I parametri da passare a tale funzione sono costituiti da una struttura contenente due campi, `mode` e `tick`.

Il campo `mode` indica la modalità con cui deve essere programmato il real-time clock.

- Se tale valore vale `LL_PERIODIC` il realtime clock viene programmato in modo *periodico*, cioè ogni `tick` microsecondi viene generata una interruzione di timer. Ad ogni interruzione il sistema gestisce eventuali eventi la cui attivazione è minore del tempo corrente (vedi Figura 5.4). Uno dei problemi dell'approccio periodico è che la gestione degli eventi effettuata in seguito alla generazione di un interrupt non può durare più di `tick` $\mu s$.

- Se tale valore è diverso da `LL_PERIODIC` il realtime clock viene pro-

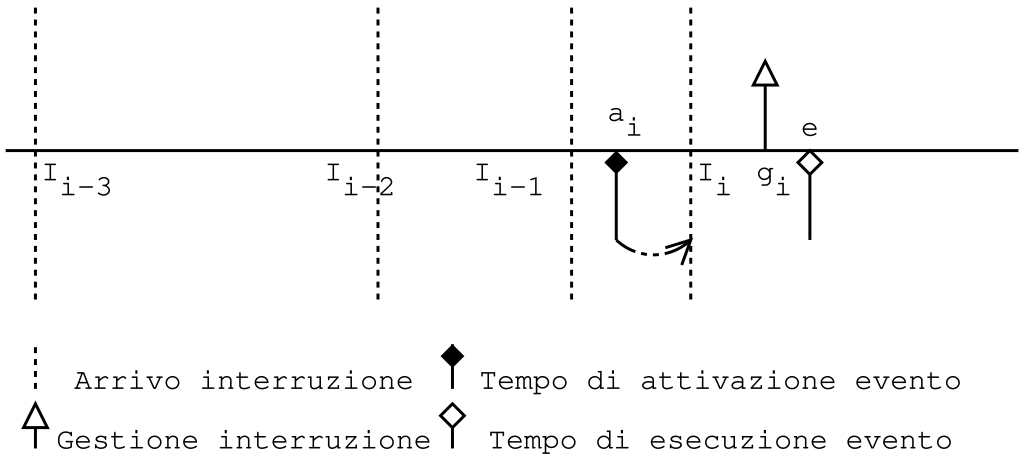


Figure 5.5: Clock real-time programmato in modo *one shot*. Gli interrupt vengono generati in corrispondenza del tempo di attivazione dell'evento. Il tempo che passa tra $a$ ed $I_i$ è dovuto al fatto che due interrupt consecutivi fisicamente non possono essere generati a distanza minore di circa 50 $\mu sec$ (questo limite dipende da architettura ad architettura ed è stabilito dalla OS Lib); tipicamente tale ritardo è **assente**. Il tempo che passa tra $I_i$ e $g_i$ è un ritardo dovuto ad eventuali disabilitazioni delle interruzioni al momento dell'arrivo dell'interrupt. Il tempo che passa tra $g_i$ ed $e$ è dovuto in generale alla esecuzione di altri eventi durante la stessa interruzione; tipicamente tale ritardo è **assente**.

grammato in modo *one shot*, ovvero viene generata una interruzione solo quando necessario (vedi Figura 5.5) . L'implementazione del modo di programmazione one shot prevede internamente l'utilizzo anche del timer collegato allo speacker di sistema, pertanto in modalità one-shot non è possibile utilizzare tale periferica. Non sono presenti limiti di tempo per l'esecuzione degli eventi in quanto la OS Lib aggira con alcuni trucchi il caso in cui un evento sia generato oltre il tempo massimo supportato dalla interfaccia di un PC (poco più di 50 ms).

## 5.8 Funzioni di terminazione della OS Lib

Prima che termini una applicazione scritta utilizzando la OS Lib è necessario rilasciare le risorse e le strutture dati allocate dalla OS Lib. Tale rilascio può essere effettuato tramite due funzioni, chiamate `ll_end` ed `ll_abort`, i cui prototipi sono presentati in Figura 5.3.

La funzione `ll_end` viene richiamata quando il sistema può effettuare uno shutdown in modo corretto. Tale funzione *deve* essere richiamata all'interno del contesto di partenza del sistema.

La funzione `ll_abort` permette di uscire in modo "pulito" dal sistema anche nei casi più drammatici. La funzione non ha bisogno di trovarsi all'interno del contesto di partenza, e la sua esecuzione provoca l'uscita incondizionata del sistema utilizzando uno stack di "salvataggio" appositamente predisposto. La funzione permette inoltre di restituire un codice di errore.

## 5.9 Inizializzazione del sistema

In questo paragrafo verranno affrontate le problematiche riguardanti la inizializzazione del sistema per quanto riguarda una applicazione che utilizzi la OS Lib. In particolare, verranno evidenziate le funzioni chiamate e da chiamare affinchè il sistema si inizializzi e termini nel modo corretto.

Innanzitutto è utile chiarire l'approccio seguito per lo sviluppo di una applicazione con la OS Lib. La OS Lib non è un sistema a sé stante, ma, come dice il nome, si tratta di un insieme di librerie di supporto alla implementazione di sistemi operativi. In questa ottica una applicazione (ad esempio un Kernel) che utilizzi la OS Lib verrà prodotta linkando le parti scritte dall'utente finale assieme alle librerie fornite, per formare una immagine binaria eseguibile.

L'immagine binaria prodotta utilizzando la OS Lib segue uno standard chiamato *multiboot*[12]; tale immagine può venire caricata in memoria tramite un boot-loader compatibile con lo standard come GRUB [Bol]. In alternativa ai boot-loader la OS-Lib mette a disposizione un *extender* proprietario, cioè un programma MS-DOS che si preoccupa di caricare in memoria e far partire l'immagine binaria della applicazione.

Una volta che l'immagine binaria è stata caricata in memoria viene lanciata una funzione interna della OS Lib, che si preoccupa di far passare in modo protetto la CPU e di riempire la struttura multiboot da passare all'applicazione. Fatto questo, la funzione interna richiama l'applicazione vera e propria.

L'applicazione inizia con una funzione chiamata `__kernel_init__`[13], a cui viene passata come parametro una struttura multiboot. La struttura di tale funzione è costituita da una serie di passi esemplificati nei seguenti punti fondamentali:

1. Inizializzazione del sistema;

2. Esecuzione della applicazione vera e propria;

3. Terminazione del sistema.

La inizializzazione del sistema si compone tipicamente di tre passi fondamentali:

- La prima cosa da fare è in generale predisporre le strutture dati interne per portarle in uno stato consistente (ad esempio, per un Kernel, è necessario inizializzare i descrittori e le code dei processi).

[12] In questo documento non entreremo nel dettaglio dello standard multiboot. La cosa importante da conoscere é che esso é uno standard che specifica la struttura binaria delle immagini da caricare e permette di passare alle applicazioni delle informazioni riguardanti l'architettura della macchina, quali, ad esempio, la memoria disponibile, eventuali parametri passati a riga di comando, ecc...

[13] il prototipo di tale funzione è `void __kernel_init__(struct multiboot_info *multiboot);`

- Come secondo passo[14] occorre inizializzare un gestore della memoria libera[15] utilizzando le informazioni presenti nella struttura multiboot.

- Infine occorre inizializzare la OS Lib vera e propria, chiamando le funzioni `ll_init` ed `event_init`.

La terminazione del sistema corrisponde alla chiamata delle funzioni di terminazione della OS Lib, quali `ll_end`, seguita dalla terminazione naturale della funzione `__kernel_init__`, oppure da una chiamata alla funzione `ll_abort`.

[14] Il secondo passo sarà presente solo se la Applicazione da eseguire necessita di un allocatore dinamico della memoria...

[15] La OS Lib non fornisce allocatori di memoria, che devono pertanto essere implementati dai livelli superiori.

# Capitolo 6

# I modelli

In questo capitolo verranno descritte nel dettaglio le definizioni e la metodologia usata riguardanti i Modelli utilizzati all'interno del Kernel. Per una descrizione generale sui Modelli di rimanda alla Sezione 3.3.

## 6.1 La metodologia utilizzata nella definizione delle strutture dati

Uno degli obiettivi dello sviluppo del Kernel, come più volte è stato ripetuto nei capitoli precedenti, è quello della modularità. Uno dei punti cardini dell'implementazione della modularità a tutti i livelli riguarda l'indipendenza delle strutture dati utilizzate nel Kernel Generico dai possibili approcci perseguiti all'interno dei Moduli.

Per questo motivo particolare cura è stata proffusa affinchè le strutture dati utilizzate agevolassero la indipendenza del codice.

Fondamentalmente i problemi da risolvere sono stati i seguenti:

- Creare una definizione di base delle seguenti strutture dati:
  - Modelli di task

- Modelli di Risorse
- Modelli di Mutex
- Descrittori di Task
- Descrittori di Moduli di Schedulazione
- Descrittori di Moduli di Gestione delle Risorse
- Descrittori di Moduli che implementano Protocolli di Accesso a Risorse Condivise

- Definire metodi con cui poter creare estensioni alle strutture dati dei punti precedenti [1]. Le estensioni dovrebbero seguire le seguenti due linee guida
  - Una estensione deve poter essere realizzata in modo indipendente dal resto del codice scritto, cioè deve essere possibile aggiungere nuovi modelli o estendere quelli presenti senza ricompilare il vecchio codice[2]
  - Le estensioni devono nei limiti del possibile non coinvolgere strutture dati allocate dinamicamente, in quanto spesso l'allocazione dinamica dei dati ha un overhead non predicibile, pertanto poco idoneo ad un sistema real-time.

- Mantenere il codice il più leggibile e comprensibile possibile, per facilitarne la manutensione e l'estensione futura

Ovviamente esistono moltissimi approcci con cui risolvere il problema delle estensioni. Di seguito ne saranno proposte alcune di quelle scartate per le motivazioni esposte. Infine sarà presentata la soluzione realmente adottata.

[1] Le estensioni in generale sono necessarie per implementare nuovi algoritmi in nuovi Moduli

[2] ciò rende ad esempio possibile una distribuzione del kernel in forma precompilata...

Un primo metodo per implementare le estensioni è ad esempio quello di implementare la parte comune di una struttura dati in una `struct` e le estensioni attraverso una `union` contenuta all'interno della `struct`. Questo approccio costringe però il sistema a non essere più indipendente dalle estensioni create, in quanto una nuova estensione deve essere inclusa all'interno della `union` che così diventa una `union` *gigante* poco leggibile e manutenibile. Inoltre il compilatore non diviene più capace di trovare errori quali l'utilizzo di variabili appartenenti a parti alternative di tale struttura dati.

Un altro metodo preso in considerazione è quello che prevede di inserire, all'interno della `struct` contenente i dati comuni, un parametro di tipo `void *` che punti alla estensione "valida" in un dato momento. Se da un punto di vista tale approccio mantiene indipendenti le estensioni dal Kernel Generico, rimane il problema della allocazione dinamica delle estensioni, che in ogni caso rende il codice scritto più lungo e meno comprensibile.

L'approccio effettivamente implementato è invece un approccio simile a quelli che si possono ritrovare all'interno dei linguaggi ad oggetti: in pratica il Kernel Generico definisce una vera e propria *classe base*, ovvero una `struct` contenente le informazioni comuni a tutte le istanze del modello.

Chiunque abbia la necessità di estendere la classe base creando una *classe derivata*, crea una nuova struttura, nellla quale il primo parametro è una `struct` del tipo della classe base[3], ed i restanti campi sono le eventuali estensioni a tale classe.

Al momento di passare una classe derivata ad una funzione che richiede un puntatore alla classe base, viene effettuato un cast esplicito dalla classe derivata alla classe base. Essendo il primo membro della classe derivata un oggetto di tipo "classe base", il cast esplicito restituisce effettivamente un puntatore corretto, in quanto le strutture vengono allocate seuendo l'ordine lessicale dei membri all'interno della dichiasrazione.

Viceversa un Modulo, quando riceverà un puntatore alla classe base, po-

[3]"ereditando" in questo modo i campi comuni

trà, dopo averne controllato la correttezza, effettuare un cast dalla classe base alla classe derivata, "recuperando" in questo modo le estensioni apportate.

Questo modo di operare può essere interpretato come una implementazione della ereditarietà singola e del polimorfismo all'interno di un linguaggio non ad oggetti quale il C.

Esso inoltre soddisfa tutti gli obiettivi posti all'inizio del paragrafo[4], e pertanto è stato scelto come metodo effettivo con cui vengono implementate tutte le strutture dati estendibili all'interno del Kernel Generico.

## 6.2 Modelli di Task

I Modelli di Task sono implementati, come descritto nel paragrafo precedente, tramite estensioni di una struttura base. Di seguito è presentata la definizione di tale classe base e la descrizione delle principali estensioni ad essa applicate.

I Modelli di Task contenuti nella distribuzione ufficiale del Kernel sono inclusi nel file `include/kernel/model.h`.

### 6.2.1 `TASK_MODEL`

Questa struttura rappresenta la struttura di base per i Modelli di Task. La sua definizione è presentata in Figura 6.1. Di seguito sono descritti i significati di ciascun campo della struttura.

`pclass` rappresenta una vera e propria indicazione di tipo con la quale i moduli possono capire a quale classe derivata si sta facendo riferimento. Le `pclass` delle classi derivate distribuite ufficialmente con il Kernel sono incluse nel file `include/kernel/model.h`. Per gli altri codici fare riferimento a (COME SCRIVERE I MODULI). Le `pclass` sono contenute all'interno di un intero a 16 bit, organizzato nel seguente modo:

[4] difficile da credere, ma soddisfa anche il requisito della leggibilità. Una volta compreso infatti il significato dei cast espliciti, il codice aggiunto per implementare il polimorfismo é poco e posizionato sempre in un punto preciso, all'inizio delle funzioni.

```
typedef struct {
    WORD pclass;
    size_t stacksize;
    void *stackaddr;
    WORD group;
    void *arg;
    DWORD control;
} TASK_MODEL;
```

Figura 6.1: la `struct TASK_MODEL`.

- Gli 8 bit più significativi forniscono una indicazione sul tipo effettivo della estensione.
- Gli 8 bit meno significativi servono per indicare il livello a cui si riferisce in modo "esclusivo" il modello[5].

`stacksize` rappresenta la dimensione in byte richiesta per lo stack del task da creare. Il Kernel Generico si preoccupa per ogni task creato (a meno che non venga specificato il parametro `stackaddr`), di allocare e deallocare quando necessario stacksize bytes di memoria da utilizzare come stack.

`stackaddr` rappresenta un puntatore ad una locazione di memoria da utilizzare come stack. Se specificato, il parametro `stacksize` viene ignorato al momento della creazione del task. La memoria puntata da `stackaddr` non viene deallocata alla terminazione del task relativo (tale memoria se necessario deve essere deallocata a cura dell'utente che

[5] Occorre ricordare innanzitutto che i Moduli di Schedulazione sono organizzati a livelli. I Moduli di Task vengono utilizzati al momento di creare i task e vengono passati ai vari Moduli alla ricerca di un Modulo che riesca a gestire un dato Modello. I Moduli in generale accettano Modelli in cui gli 8 bit meno significativi del campo `pclass` valgono 0 oppure contengono il numero di livello effettivo in cui il modulo é stato registrato. In questo modo é possibile distinguere Modelli di Task destinati a Moduli diversi nel caso in cui ci siano più Moduli che accettano una stessa classe derivata.

crea il task). Non viene effettuato alcun controllo sulla dimensione dell'area di memoria puntata da `stackaddr`.

`group` i task all'interno del sistema sono divisi in gruppi. Il gruppo di appartenenza di un task viene specificato al momento della creazione di un task utilizzando questo campo. Un gruppo è un identificatore numerico che caratterizza un insieme di task all'interno del sistema. Tale identificatore viene utilizzato nelle primitive group_activate e group_kill per richiedere in modo atomico l'attivazione e la terminazione di un insieme di task (vedi Sezioni 10.4 e 10.12).

`arg` il corpo di ogni task creato all'interno del sistema è una funzione che accetta un parametro di tipo `void *`. Il parametro che viene effettivamente passato alla prima attivazione di un task viene specificato all'interno di questo campo.

`control` questo campo contiene un insieme di bits che rappresentano funzionalità riguardanti i task creati e/o da creare. Di seguito sono descritti per completezza tutti i flag definiti per il campo control del `TASK_MODEL` e per il campo control del descrittore di Task. Tali flag possono venire settati dall'utente alla creazione di un Modello di Task oppure vengono settati tramite apposite primitive di sistema. I flag definiti sono i seguenti:

  `USE_FPU` Questo flag è utilizzato dalla OS Lib per indicare che il task in oggetto può utilizzare all'interno del proprio codice delle operazioni in virgola mobile. Questa dichiarazione risulta necessaria in quanto non è detto che la CPU effettui automaticamente il salvataggio dei registri interni del coprocessore matematico ad ogni context switch. Se specificato, la OS Lib si preoccupa di salvare il contesto del coprocessore matematico quando necessario.

NO_KILL Se questo flag è settato il task non può venire ucciso tramite la primitiva task_kill. Questo flag viene utilizzato inoltre al momento dello shutdown del Kernel (vedi 10.1), e può essere specificato solo alla creazione di un task.

NO_PREEMPT Se questo flag è settato il task non può subire preemption. In altre parole, lo scheduler viene disabilitato. Questo flag viene modificato dalle primitive task_preempt e task_nopreempt.

SYSTEM_TASK Questo flag indica se il task è un task di sistema o meno. Esso può essere specificato solo alla creazione di un task. Il fatto che un task sia o meno di sistema influenza lo shutdown di sistema (vedi 10.1).

JET_ENABLED Questo flag può essere settato solo al momento della creazione di un task ed indica se il Kernel Generico debba o meno registrare delle statistiche inerenti i tempi di esecuzione dei task.

TASK_JOINABLE Questo flag indica se il task può subire una attesa di terminazione tramite la primitiva di join messa a disposizione dal Kernel Generico. per ulteriori informazioni fare riferimento a (vedi 10.9).

STACKADDR_SPECIFIED Questo flag indica se alla creazione di un task è stato o meno specificato un address per lo stack. In caso negativo, alla fine del task l'area di memoria precedentemente allocata dal Kernel Generico per contenere lo stack va liberata.

TRACE_TASK Questo flag indica se il tracer debba o meno monitorare gli eventi di un task. Per una documentazione sul tracer fare riferimento a [Gio00].

KILLED_ON_CONDITION Questo flag è utilizzato all'interno della implementazione delle wait sulle variabili condition. Il motivo dell'esistenza di tale flag è che un task ucciso mentre attende all'interno di una condition wait deve essere rischedulato per riacquisire il mutex associato alla variabile condition prima di morire.

`KILL_ENABLED` Questo flag indica se la cancellazione (in senso POSIX) è abilitata o meno. Per maggiori informazioni fare riferimento alla Sezione 10.7.

`KILL_DEFERRED` Questo flag indica se la cancellazione (in senso POSIX) è differita o asincrona (flag resettato). Per maggiori informazioni fare riferimento alla Sezione 10.7.

`KILL_REQUEST` Questo flag registra una richiesta di cancellazione. Le richieste di cancellazione vengono fatte tramite primitive quali `task_kill`, `group_kill`, `pthread_testcancel`.

`CONTROL_CAP` Questo flag è utilizzato internamente ai Moduli di Schedulazione ed indica se il Kernel Generico debba o meno generare automaticamente un evento di fine capacità per un task appena schedulato.

`TASK_DOING_SIGNALS` Questo flag viene settato quando un task è all'interno di un handler di un segnale POSIX ed è utilizzato solo all'interno della funzione `kern_deliver_pending_signals`, contenuta nel file `kernel/signal.c`.

`FREEZE_ACTIVATION` Questo flag blocca l'effetto di una attivazione effettuata tramite le primitive task_activate o group_activate. Il flag è modificato dalla primitive `task_block_activation` e dalla primitiva `task_unblock_activation`.

`WAIT_FOR_JOIN` Questo flag viene settato al momento della terminazione di un task se il task è di tipo joinable. Il flag indica che il task è terminato e sta aspettando che qualcuno effettui una `task_join` per poter morire definitivamente. Per maggiori informazioni fare riferimento alla Sezione 10.9.

DESCRIPTOR_DISCARDED Questo flag è utilizzato nella implementazione della task_join, e viene settato dalla primitiva task_create per indicare che un descrittore di task, seppur occupato in attesa di una task_join, è stato scelto e scartato per la creazione di un nuovo task. Per maggiori informazioni fare riferimento alla Sezione 10.9.

SIGTIMEOUT_EXPIRED Questo flag viene settato per i task bloccati su una sigtimedwait quando il timeout effettivamente termina. Per maggiori informazioni fare riferimento alla Sezione 10.8.

Le strutture dati interne ad una variabile TASK_MODEL dovrebbero essere utilizzate solo all'interno del Kernel Generico e dei Moduli. Per questo motivo il sistema mette a disposizione una serie di macro utilizzabili per impostare i valori richiesti all'interno di un oggetto della classe base[6]. In tutte le macro descritte di seguito il parametro m è un TASK_MODEL Le macro a disposizione sono pertanto le seguenti:

task_default_model(m,p) Questa macro inizializza m a valori di default. La pclass di m diventan uguale al parametro p.

task_def_level(m,l) Questa macro specifica il livello l di schedulazione a cui riferimento il Modello m.

task_def_arg(m,a) Questa macro serve per specificare il parametro passato alla prima attivazione dei task inizializzati con il Modello m.

task_def_stack(m,s) Questa macro serve per specificare la dimensione dello stack dei task inizializzati con il Modello m.

[6]Si anticipa che tali macro non verranno richiamate direttamente dall'utente ma verranno richiamate tramite un livello di indirezione, passando cioé tramite le classi derivate.

`task_def_stackaddr(m,s)` Questa macro serve per specificare un puntatore ad una zona di memoria da utilizzare come stack per i task inizializzati con il Modello `m`.

`task_def_group(m,g)` Questa macro serve per specificare gruppo dei task inizializzati con il Modello `m`.

`task_def_usemath(m)` Questa macro serve per specificare che i task inizializzati con il Modello `m` utilizzano il coprocessore matematico.

`task_def_system(m)` Questa macro serve per specificare che i task inizializzati con il Modello `m` sono task di sistema.

`task_def_nokill(m)` Questa macro serve per specificare che i task inizializzati con il Modello `m` non possono essere uccisi

`task_def_ctrl_jet(m)` Questa macro serve per specificare che i task inizializzati con il Modello `m` richiedono il monitoraggio dei tempi di esecuzione.

`task_def_joinable(m)` Questa macro serve per specificare che i task inizializzati con il Modello `m` sono task che possono essere indicati all'interno di una primitiva `task_join`.

`task_def_unjoinable(m)` Questa macro serve per specificare che i task inizializzati con il Modello `m` sono task che non possono essere indicati all'interno di una primitiva `task_join`.

`task_def_trace(m)` Questa macro serve per specificare che i task inizializzati con il Modello `m` sono task per i quali il tracer di sistema deve registrare delle informazioni.

`task_def_notrace(m)` Questa macro serve per specificare che i task inizializzati con il Modello `m` sono task per i quali il tracer di sistema non deve registrare nessuna informazione.

### 6.2.2 Convenzioni utilizzate

Nella pratica l'utente finale non dovrà mai istanziare un oggetto di classe `TASK_MODEL`, ma un oggetto da lui derivato. Anche le macro descritte nel paragrafo precedente non dovranno essere utilizzate direttamente; al loro posto saranno predisposte macro analoghe.

Le convenzioni utilizzate nella derivazione di un Modello di Task dal `TASK_MODEL` sono espresse nei seguenti punti:

1. Il nome di una classe derivata si ottiene dal nome della classe base aggiungendo un prefisso. Per esempio, il Modello di Task per i task soft si chiama `SOFT_TASK_MODEL`.

2. Il primo elemento della struttura dati della classe derivata è una struttura `TASK_MODEL` chiamata `t`.

3. I parametri specifici aggiunti dal nuovo Modello (ad esempio periodo, wcet, ecc...) sono inseriti come normali membri della struttura di seguito al primo parametro. Tali parametri aggiuntivi devono essere interpretati come una specifica di Qualità del Servizio richiesta da un Task al sistema al momento della propria creazione. I Moduli di Schedulazione possono anche essere strutturati in modo tale da utilizzare solo parte di tali parametri[7]. Questo approccio evita il proliferare dei Modelli di Task ed ha come vantaggio una maggiore indipendenza delle Applicazioni dal Modello di Task effettivamente utilizzato[8].

[7] Ad esempio, per creare un modello di task soft può essere utile inserire due parametri quali il periodo ed il mean execution time; tali parametri saranno utilizzati (e richiesti obbligatoriamente) da un Modulo di Schedulazione che implementi l'algoritmo CBS, mentre potranno essere ignorati da un Modulo di Schedulazione che implementa un Polling Server.

[8] Facendo riferimento alla nota precedente, una applicazione potrebbe utilizzare un Modello di Task soft specificando tutti i parametri del modello, senza preoccuparsi poi del fatto che il proprio task sia schedulato con un algoritmo CBS o Polling Server...

4. Le macro fornite con il `TASK_MODEL` e descritte nel paragrafo precedente devono essere fornite anche per il nuovo modello. Esse dovrebbero essere riscritte con il prefisso aggiunto per la struttura di base, in modo simile al seguente esempio:

```
#define soft_task_def_level(m,l) task_def_level((m).t,l)
```

5. Dovrebbero essere fornite un insieme di macro simili a quelle fornite con il `TASK_MODEL` per la manipolazione dei nuovi campi della struttura derivata. Ad esempio, se il nuovo modello per i task soft ha al suo interno un campo `period` (contenente ad esempio un periodo di riattivazione), la macro da fornire potrebbe essere simile alla seguente:

```
#define soft_task_def_period(m,p) (m).period = (p)
```

Le cinque regole esposte sopra sono fondamentalmente regole di buon senso che permettono di espandere in modo semplice ed omogeneo i Modelli di Task forniti con il sistema. Esse inoltre semplificano la scrittura di nuovi modelli, in quanto spesso e volentieri è sufficiente modificare i prefissi delle macro e poco altro per avere una nuova classe derivata.

### 6.2.3 Modelli di Task integrati all'interno del Kernel

Attualmente il Kernel definisce un insieme di Modelli di Task direttamente derivati dal TASK_MODEL. Questo insieme di Modelli derivati viene poi utilizzato all'interno dei Moduli di Schedulazione distribuiti con il Kernel Generico. Particolare attenzione è stata posta nella scelta dei campi di tali Modelli, in modo tale da diminuirne, dove possibile, il numero totale. Di seguito verranno descritte brevemente le caratteristiche di tali Modelli; Non verranno indicate per brevità le definizioni delle strutture dati e delle macro correlate (fare riferimento al file `include/kernel/model.h` per tali informazioni).

`HARD_TASK_MODEL`

Questo Modello di Task può essere utilizzato per modellare i task Hard periodici e sporadici.

Con Task Hard Periodico[9] si intende un task garantito utilizzando un periodo di attivazione ed un wcet[10], mentre un task Hard Sporadico[11] è un task aperiodico garantito basandosi su di un minimo tempo di interarrivo ed un wcet.

In entrambe i casi può essere specificata una deadline relativa. Se non specificata, si assume che la deadline di un task coincida con il successivo istante di riattivazione.

In generale un Modulo di Schedulazione che accetta task Hard dovrebbe poter generare le seguenti eccezioni:

`XDEADLINE_MISS` Se il task hard supera una deadline senza aver terminato correttamente l'istanza.

`XWCET_VIOLATION` Se il task tenta di consumare più tempo di quanto dichiarato.

`XACTIVATION` Se in generale si tenta di attivare un task *troppo spesso*.

`SOFT_TASK_MODEL`

Questo modello viene tipicamente utilizzato per modellare i task soft. Con task soft si intendono dei task, periodici o aperiodici, usualmente non garantiti o garantiti basandosi su di un periodo ed un tempo medio di esecuzione (`met` - mean execution time).

Un task soft inoltre può specificare, tramite un apposito campo, se intende o meno registrare *attivazioni pendenti*. Con il concetto di attivazione pendente si intende il fatto che un task può essere riattivato

[9] Il campo `periodicity` del modello deve valere `PERIODIC`

[10] Worst Case Execution Time.

[11] Il campo `periodicity` del modello deve valere `APERIODIC`

prima ancora che finisca la propria attivazione corrente[12]. I Moduli di Schedulazione possono scegliere se gestire o meno tale caratteristica. nel caso in cui la gestiscano essa influenza il comportamento delle primitive `task_sleep` e `task_endcycle` (vedi 8.3.2).

Il Modello dispone anche di una indicazione relativa ad un wcet, per permettere a Moduli che richiedono tale grandezza (ad esempio, quelli derivati dall'algoritmo TBS) di utilizzare un Modello di Task soft.

I task soft tipicamente non generano nessun tipo di eccezione.

`NRT_TASK_MODEL`

Questo Modello di Task è tipicamente utilizzato per supportare computazioni di task non real-time, cioè task che tipicamente non hanno requisiti temporali riguardanti la propria esecuzione, o per supportare task real-time per i quali non viene realizzata garanzia on-line.

Tipici Moduli di Schedulazione che utilizzano tale modello sono i Moduli che implementano algoritmi quali Round Robin, Proportional Share, task prioritari, nonchè lo scheduler specificato dallo standard POSIX.

Il modello dispone anche di un campo che permette di specificare se il Task debba o meno salvare le attivazioni pendenti.

Infine, il modulo dispone di altri due campi utilizzati all'interno dello scheduler POSIX[13].

`JOB_TASK_MODEL`

Questo Modello di task è utilizzato normalmente come Modello passato dai Server Aperiodici verso i rispettivi Moduli Master, e pertanto non è tipicamente utilizzato direttamente nel codice delle Applicazioni.

[12] Questo fatto può verificarsi a causa della natura della garanzia effettuata sul task. Infatti, se un task garantito con proprio tempo medio esegue più del dovuto, la terminazione della istanza può venire posticipata, dando modo ad altri task di attivarlo nuovamente.

[13] Questa è una delle poche eccezioni nel sistema all'indipendenza tra Modello e Modulo di Schedulazione... (nulla è perfetto, se la perfezione esiste)

Tale Modello aggiunge al Modello base unicamente una deadline ed un periodo. Con Job si intenede infatti una singola attivazione di un task, che comincia e finisce senza sincronizzazioni intermedie.

Tipicamente un server aperiodico inserisce un Job nel Modulo Master per richiedere la schedulazione di un proprio task. Nel momento in cui tale task termina una istanza o si blocca su di una sincronizzazione, il Job viene fatto direttamente morire per poi ricrearlo ex-novo al momento della riattivazione o all'uscita della sincronizzazione.

La mancanza di campi per la specifica dei tempi di calcolo (ad esempio campi quali wcet, met, ecc...) è motivata dal fatto che il Server Aperiodico che utilizza tali Modelli generalmente deve preoccuparsi in modo dedicato della gestione della capacità.

Notare come, se consideriamo una architettura classica basata su EDF o RM, creare/far morire un Job con una certa deadline e periodo equivale ad inserire/rimuovere un task in coda pronti.

Il Modello contiene infine un ulteriore flag che serve per specificare se il Modulo Master debba o meno generare una eccezione di deadline miss nel caso in cui il Job sia ancora presente nel sistema al raggiungimento del tempo indicato nella deadline.

## 6.3 Modelli di Risorse

I Modelli di Risorse sono implementati in modo simile ai Modelli di Task. Le differenze con tali Modelli sono in questo caso dovute al fatto che non è possibile stabilire una serie di parametri comuni a tutti i possibili Gestori di Risorse. Per questo motivo la struttura che rappresenta la classe base contiene solo il campo che fornisce informazioni sul reale tipo del Modello di Risorsa. Analogamente a quanto succede per i Modelli di Task, tale campo è chiamato `rclass`.

I Modelli di Risorsa che sono stati effettivamente implementati sono utilizzati nell'implementazione dei protocolli Priority Ceiling ed SRP.

Per quanto riguarda il protocollo Priority Ceiling, il Modello di Risorsa utilizzato aggiunge al Modello base la priorità statica del task da creare.

Per quanto riguarda invece il protocollo SRP, il Modello di Risorsa utilizzato aggiunge al Modello base la definizione del livello di preemption del task.

I Modelli di Risorse contenuti nella distribuzione ufficiale del Kernel sono inclusi nel file `include/kernel/model.h`.

## 6.4 Attributi dei Mutex

Come accennato nella Sezione 4.4.2, il protocollo effettivamente utilizzato da un mutex all'interno del sistema viene stabilito a runtime al momento di inizializzare il mutex stesso.

Per implementare tale approccio in modo modulare si è operato in modo simile a quanto fatto per i Modelli di Risorse: a partire da una classe base (in questo caso chiamata `mutexattr_t`) sono stati derivati una serie di Attributi di Mutex da utilizzare al momento della inizializzazione delle strutture `mutex_t`.

L'approccio seguito nel naming delle strutture è identico a quello seguito nei Modelli descritti nei paragrafi precedenti, pertanto non verrà qui ripetuto ulteriormente.

Gli Attributi di Mutex contenuti nella distribuzione ufficiale del Kernel sono inclusi nel file `include/kernel/model.h`.

# Capitolo 7

# Supporti forniti agli sviluppatori di Moduli

Il Kernel Generico attualmente non implementa alcun tipo di protezione della memoria. Pertanto, un utilizzatore che sviluppa un Modulo di Schedulazione e/o una Applicazione ha la visibilità completa di molte strutture dati interne al Kernel Generico.

Il fatto in ogni caso che una struttura dati o una variabile sia visibile in un determinato modulo non significa che essa possa tranquillamente essere letta o modificata[1].

## 7.1 Tipi base definiti dal Kernel

Il Kernel Generico utilizza al suo interno una serie di tipi primitivi. Nel seguito della Sezione verranno descritti tali tipi assieme ad alcune considerazioni

[1] Una corretta progettazione pretende giustamente che un modulo esporti solo i simboli utilizzabili dalle entità esterne. Di fatto questo modo di operare porta ad una maggiore chiarezza, ma per ragioni storiche e di comodità non è stato realizzato ovunque. Può capitare quindi di avere accessibili in un modulo non solo i simboli utilizzabili, ma anche altri simboli "pericolosi"... sta alla bontà del programmatore utilizzare gli strumenti messi a disposizione nel modo corretto.

di utilità pratica.

**PID** viene utilizzato per contenere un indice di Task all'interno del sistema. Esso è un intero ed indica una posizione all'interno della tabella dei descrittori di task. I valori significativi vanno da 0 a MAXPROC-1 oltre all'identificatore di task non valido, NIL.

**QUEUE, QQUEUE** vengono utilizzati per implementare le code di task utilizzate tipicamente negli algoritmi di schedulazione. Fare riferimento alla Sezione 7.6.5.

**TASK** questo tipo è semplicemente una ridefinizione del tipo `void *` e può essere utilizzato a scopi di leggibilità nella dichiarazione dei task come tipo ritornato dalle funzioni.

**LEVEL, RLEVEL** vengono utilizzati per indicare un Modulo rispettivamente di Schedulazione o di Gestione delle Risorse. Essi sono interi ed indicano una posizione all'interno della tabella dei descrittori di riferimento (vedi Sezione 7.6.5). I valori significativi vanno da `0` a `MAX_SCHED_LEVEL-1` (per il tipo `LEVEL`) o `MAX_RES_LEVEL` (per il tipo `RLEVEL`).

**bandwidth_t** questo tipo viene utilizzato per mantenere un numero reale compreso tra 0 ed 1. In realtà il tipo è implementato come un intero unsigned a 32 bit, ed il valore x contenuto va interpretato come $\frac{x}{MAX\_BANDWIDTH}$.

**task_key_t** questo tipo è un intero e viene utilizzato come indice per i dati specifici dei task (l'equivalente dei thread specific data dello standard POSIX)

## 7.2 Descrittori

Le strutture dati utilizzate più spesso all'interno del Kernel Generico e dei Moduli sono sicuramente i descrittori di task, di livello e di risorsa. In questa sezione saranno presentati tali descrittori, descrivendone in dettaglio tutti i campi. Se non diversamente specificato, le strutture dati descritte sono dichiarate nel file `include/kernel/descr.h` e sono definite (di solito come elementi di array) nel file `kernel/kern`.c.

### 7.2.1 Descrittore di Task

I descrittori di task sono dichiarati nella struttura dati `proc_des`[2]. Di seguito viene presentata una descrizione dei vari campi di tale struttura, assieme al loro tipo.

**DWORD task_ID** Numero progressivo associato al task al momento della creazione.

**LEVEL task_level** Livello reale di appartenenza del Task. Le operazioni Locali (Task Calls) di un task (vedi Sezione 4.2.4) vengono chiamate sul Modulo indicato da tale campo del descrittore di task.

**CONTEXT context** Questo campo contiene il contesto assegnato ad un determinato task.

**BYTE *stack** Questo campo contiene il puntatore all'area di memoria utilizzata dal task come stack.

**TASK (*body)()** campo contiene il puntatore al corpo del task, ovvero il puntatore alla prima istruzione del task.

[2]Il nome è stato mantenuto per ragioni storiche identico alle versioni precedenti di Hartik. Si tratta in ogni caso di un descrittore di *task* e non di *processo*.

**char name[MAX_TASKNAME]** Nome simbolico del task, la cui lunghezza massima è definita nella costante `MAX_TASKNAME`.

**WORD status** Stato del task.

**WORD pclass** Codice di classe utilizzato dal Modello di Task passato alla creazione del task. Può essere utilizzato dai Moduli per sapere la tipologia del task in questione, utile se un Modulo supporta diversi Modelli di Task.

**WORD group** Gruppo del task. Il valore 0 viene utilizzato per rappresentare la non appartenenza ad un gruppo (task singolo).

**WORD stacksize** Dimensione dell'area di stack riservata per il task.

**DWORD control** Flags di stato del task. La descrizione di tali flags è contenuta nella Sezione 6.2.1.

**int frozen_activations** Numero di attivazioni congelate. Significativo solo se il flag `FREEZE_ACTIVATION` del campo `control` è attivato.

**int sigmask** Maschera dei segnali bloccati dal task.

**int sigpending** Maschera dei segnali pendenti sul task e non ancora gestiti.

**int sigwaiting** Maschera di attesa per i segnali specificati nelle primitive `sigwait` o simili.

**struct timespec request_time** Tempo in cui è avvenuta l'ultima attivazione del task, utile per statistiche.

**int avail_time** Campo contenente il tempo di calcolo disponibile per il task (vedi Sezione 7.5).

**PID shadow** Puntatore shadow (vedi Sezione 4.4.5).

**struct _task_handler_rec *cleanup_stack** Puntatore al primo elemento dello stack dei cleanups (vedi Sezione 10.7.3).

**QUEUE next,prev** Puntatori utilizzati per accodare i task all'interno delle code gestite con i tipi `QUEUE` e `QQUEUE`.

**int errnumber** Codice di errore locale al task in esecuzione. Il simbolo `errno` definito nello standard C fornisce l'accesso al campo errnumber del descrittore del task in esecuzione.

**TIME jet_table[JET_TABLE_DIM]** Tabella contenente i tempi di calcolo consumati dalle ultime JET_TABLE_DIM attivazioni del task[3]. La tabella è gestita in modo circolare. Per maggiori informazioni a proposito delle stime dei tempi di calcolo dei task fare riferimento alla Sezione 10.6.

**int jet_tvalid** Numero di elementi validi all'interno della tabella `jet_table`.

**int jet_curr** Elemento corrente all'interno della tabella `jet_table`.

**TIME jet_max** Massimo tempo (in microsecondi) di esecuzione di una attivazione registrato dall'inizio della vita del task o dall'ultimo azzeramento delle statistiche.

**TIME jet_sum** Somma dei tempi di esecuzione (in microsecondi) delle attivazioni dall'inizio della vita di un task o dall'ultimo azzeramento delle statistiche. Questo campo assieme al campo `jet_n` viene utilizzato per il calcolo del tempo medio delle attivazioni del task.

**TIME jet_n** Numero di istanze su cui è stato calcolato il campo `jet_sum`.

**PID waiting_for_me** Questo campo viene settato con il valore del task che correntemente sta effettuando una `task_join` sul task corrente (il

[3] Con attivazione del task si intende una parte dell'esecuzione di un task posizionata tra due primitive di `task_endcycle` o `task_sleep`.

task che ha eseguito la `task_join` sta aspettando la morte del task considerato nel descrittore, da qui il nome del campo). Il campo vale `NIL` se nessun task ha effettuato join sul task corrente.

**void *return_value** Valore di ritorno restituito dal task alla sua morte. Tale valore viene mantenuto se il task è di tipo joinabile, in attesa che qualcuno si sincronizzi con il task (che ricordiamo, in questo caso, è già morto).

**void *keys[PTHREAD_KEYS_MAX]** Questo array contiene i dati specifici dei task (vedi Sezione 10.7.3).

**struct condition_struct *cond_waiting** Questo campo vale NULL oppure punta alla variabile condition sulla quale il task è bloccato[4].

**DWORD priority** Questo campo viene utilizzato dalle funzioni di accodamento per effettuare un accodamento ordinato. Può essere utilizzato dai Moduli anche al di là dell'utilizzo all'interno delle code[5].

**struct timespec timespec_priority** Utilizzato per gli stessi scopi del campo `priority`.

**int delay_timer** Campo utilizzato nei bloccaggi con timeout. Normalmente contiene l'indice dell'evento creato per il risveglio alla fine del sonno.

**int wcet** Questo campo è un campo di appoggio molto comodo se utilizzato in congiunzione con `avail_time`[6].

[4] Il motivo della presenza di questo campo nel descrittore è che se un task viene ucciso durante il bloccaggio su una condition wait, lo standard POSIX prevede che il task riacquisti la mutua esclusione collegata a tale condition prima di morire.

[5] La filosofia con cui i Moduli sono stati sviluppati vieterebbe in maniera assoluta la presenza di un campo di priorità all'interno del descrittore di processo. Diciamo pure che, per ragioni di efficienza, questa è la prima delle due eccezioni che confermano la regola...

[6] ...questa invece è la seconda.

### 7.2.2 Descrittore di livello

Questo descrittore contiene l'interfaccia esportata da un Modulo di Schedulazione. I Moduli dovranno *riempire* tale interfaccia completando tutti gli spazi *vuoti*, ovvero definendo tutti i puntatori a funzione definiti nell'interfaccia. Per i Moduli che non implementano una particolare funzione, si consiglia di definire comunque le funzioni in modo tale che l'unica operazione compiuta, se necessario, sia quella di generare una eccezione.

I campi definiti per il descrittore sono i seguenti:

**char level_name[MAX_LEVELNAME]** Nome simbolico del Modulo, per uso statistico. I nomi dei Moduli distribuiti con il kernel si trovano nel file `include/modules/codes.h`.

**WORD level_code** Codice numerico che identifica in modo univoco un Modulo. I codici dei Moduli distribuiti con il kernel si trovano nel file `include/modules/codes.h`.

**BYTE level_version** Codice numerico che identifica il numero di versione di Modulo. I numeri di versione dei Moduli distribuiti con il kernel si trovano nel file `include/modules/codes.h`.

**Level Calls, Task Calls e Guest Calls** La restante parte dell'interfaccia è costituita da una serie di puntatori a funzione che implementano le funzioni *virtuali* dei Moduli di Schedulazione. Tali funzioni sono ampiamente descritte nel Capitolo 8. Esse come primo parametro ricevono una variabile di tipo LEVEL utilizzabile per ritrovare il descrittore dei Moduli e quindi le strutture dati private di tali moduli[7].

[7] In effetti se si analizza l'implementazione dei linguaggi ad oggetti¸si nota come tutte le funzioni virtuali appartenenti ad una classe ricevono come primo parametro il puntatore `this`. Il primo parametro citato ha lo stesso significato.

### 7.2.3 Descrittore di Moduli di Gestione delle Risorse

Questo descrittore contiene l'interfaccia esportata da un Modulo di Gestione delle Risorse. I Moduli dovranno *riempire* tale interfaccia completando tutti gli spazi *vuoti*, ovvero definendo tutti i puntatori a funzione definiti nell'interfaccia. Per i Moduli che non implementano una particolare funzione, si consiglia di definire comunque le funzioni in modo tale che l'unica operazione compiuta, se necessario, sia quella di generare una eccezione.

I campi definiti per il descrittore sono i seguenti:

**char res_name[MAX_MODULENAME]** Nome simbolico del Modulo, per uso statistico. I nomi dei Moduli distribuiti con il kernel si trovano nel file `include/modules/codes.h`.

**WORD res_code** Codice numerico che identifica in modo univoco un Modulo. I codici dei Moduli distribuiti con il kernel si trovano nel file `include/modules/codes.h`.

**BYTE res_version** Codice numerico che identifica il numero di versione di Modulo. I numeri di versione dei Moduli distribuiti con il kernel si trovano nel file `include/modules/codes.h`.

**int rtype** Questo campo indica l'effettiva interfaccia implementata dal Modulo. Questo campo è indispensabile per implementare primitive che fanno affidamento su di una particolare interfaccia che estende quella base ma che per ragioni implementative non possono disporre subito del livello in cui si trova il modulo finale[8].

**Resource Calls** La restante parte dell'interfaccia è costituita da una serie di puntatori a funzione che implementano le funzioni *virtuali* dei Moduli di Gestione delle Risorse. Tali funzioni sono ampiamente descritte

[8] Un esempio di questo tipo di primitive è dato dalla mutex_init, in quanto al momento della inizializzazione dei mutex occorre ricercare se esiste un Modulo che implementa il protocollo specificato nell'attributo dei mutex.

| Parte | Codici |
|---|---|
| Kernel Generico | da 0 a `MODULE_STATUS_BASE`-1 |
| Algoritmi di Schedulazione | da `MODULE_STATUS_BASE` a `APER_STATUS_BASE`-1 |
| Server Aperiodici | da `APER_STATUS_BASE` a `LIB_STATUS_BASE`-1 |
| Altri (Gestione Risorse, librerie, ecc...) | da `LIB_STATUS_BASE` in poi |

Tabella 7.1: Partizionamento dei codici assegnabili al campo `status` di un task.

nel Capitolo 9. Esse come primo parametro ricevono una variabile di tipo RLEVEL (che possiede lo stesso significato del primo parametro passato alle funzioni dei Moduli di Schedulazione).

## 7.3 Stati del sistema: uso ed abuso

Come già accennato precedentemente nella Sezione 4.1.2 il concetto di *stato* di un task non ha più una connotazione "globale" per il sistema, ma una connotazione "locale".

Parlando in termini di implementazione, lo stato di un task è mantenuto in un campo all'interno del descritttore del task a cui lo stato fa riferimento.

Lo spazio dei valori di uno stato è partizionato in quattro parti, come esemplificato in Tabella 7.1.

Ciascun Modulo è tenuto (ma non obbligato) ad usare codici all'interno dei range specificati per gli stati interni dei propri task.

Questo approccio permette di gestire, seppure in modo non omogeneo, configurazioni di Moduli sufficientemente rappresentative. I Moduli del resto hanno una indicazione precisa dei codici numerici da poter utilizzare, permettendo di creare stati interni ai Moduli che pertanto divengono utili per una schematizzazione semplice degli Algoritmi.

Il fatto che due Moduli di Schedulazione utilizzino codici numerici comuni per stati diversi non provoca tipicamente nessun problema, in quanto tali codici sono utilizzati solo internamente ai moduli e dato che verranno chiamate solo le funzioni del modulo di appartenenza di un task.

La condivisione dei codici può invece risultare problematica tra Moduli di tipo diverso, ad esempio tra Moduli di Schedulazione e Moduli di Gestione delle Risorse. In questo caso è necessario porre attenzione ai codici utilizzati, in quanto se più Moduli condividono uno stesso codice non si può più essere certi dello stato di un task.

## 7.4 Variabili Globali utilizzabili all'interno dei Moduli

Le seguenti variabili sono tutte e sole[9] le variabili globali del sistema (definite nel file `include/kernel/var.h`) utilizzabili dai Moduli di Schedulazione:

**proc_des proc_table[]** Questa variabile rappresenta il vettore dei descrittori di Task. Le entrate utilizzabili sono quelle rilasciate dalla primitiva `task_create`, che si occupa di gestire la creazione dei task e quindi l'allocazione dei descrittori.

**level_des *level_table[]** Questa variabile è il vettore che contiene gli indirizzi dei descrittori dei Moduli di Schedulazione. Come è possibile notare, il vettore è un vettore di puntatori. Al momento della registrazione dei Moduli vengono allocate le entrate di questo vettore con opportuni puntatori alle estensioni del tipo `level_des`. In una ottica Object Oriented, la `level_table` può essere pensata come un array polimorfico di Moduli (la filosofia utilizzata è la stessa utilizzata per i Modelli di Task. Vedi Sezione 6.1).

[9] Sempre che per distrazione non ne sia stata omessa qualcuna...

**resource_des *resource_table[]** Questa variabile rappresenta il vettore dei descrittori dei Moduli di Gestione delle Risorse. Per questa variabile valgono gli stessi commenti fatti per la `level_table`.

**PID exec** Questa variabile contiene l'indicazione del task suggerito per l'esecuzione dai Moduli di Schedulazione. Esso può differire dal task suggerito dal valore contenuto in `exec` a causa del meccanismo degli Shadow (vedi Sezione 4.4.5). Tipicamente questa variabile è utilizzata solo nella procedura `scheduler()` interna al Kernel Generico. In ogni caso **NON** deve essere modificata, al limite solo letta per confrontarla con la variabile `exec_shadow`. La variabile può assumere valore -1 all'interno del Kernel Generico, per indicare la mancanza di un task correntemente in esecuzione (vedi Sezione 10.5).

**PID exec_shadow** Questa variabile contiene il task effettivamente in esecuzione. Esso può differire dal task suggerito dal valore contenuto in `exec_shadow` a causa del meccanismo degli Shadow (vedi Sezione 4.4.5). La variabile **NON** deve essere modificata, al limite solo letta per confrontarla con la variabile `exec`. La variabile può assumere valore -1 all'interno del Kernel Generico, per indicare la mancanza di un task correntemente in esecuzione (vedi Sezione 10.5).

**int cap_timer** Questa variabile è diversa da `-1` se il Kernel Generico ha generato un evento di fine capacità. Tale variabile può essere utilizzata per mantenere l'indice dell'evento di fine capacità creato da Moduli che non utilizzano il flag `CONTROL_CAP` per i propri task (vedi Sezione 7.5).

**struct timespec schedule_time** Questa variabile contiene il valore del tempo di sistema effettuata nell'istante di chiamata della procedura `scheduler()` (per maggiori informazioni vedi Sezione 10.5).

**struct timespec cap_lasttime** Questa variabile contiene la copia della varaibile schedule_time effettuata alla fine di ogni chiamata della procedura `scheduler()`. Essa viene utilizzata per implementare l'accounting dei tempi di calcolo (per maggiori informazioni vedi Sezioni 10.5 e 10.6).

**DWORD sched_levels** Questa variabile indica il numero di descrittori di livello realmente allocati. Tale valore viene modificato dalla funzione `level_alloc_descriptor`.

**DWORD res_levels** Questa variabile indica il numero di descrittori di risorse realmente allocati. Tale valore viene modificato dalla funzione `resource_alloc_descriptor`.

**int task_counter** Contatore per i task utente (vedi Sezione 10.2).

**int system_counter** Contatore per i task di sistema (vedi Sezione 10.2).

**int mustexit** Questa variabile è utilizzata dalle primitive di sistema `sys_end` e `sys_abort` per bloccare eventuali cambi di contesto all'interno degli event handler.

**int calling_runlevel_func** Questa variabile indica che il sistema sta eseguendo delle funzioni (indicate tramite la `sys_atrunlevel()`) di sistema. Il settaggio ad 1 di tale variabile cambia il comportamento della primitiva di sistema task_activate, in quanto in questo caso vieve chiamata all'interno del contesto globale e quindi le azioni da intraprendere sono leggermente diverse (fare riferimento direttamente al codice della primitiva, nel file `kernel/activate.c`).

**QUEUE freedesc** Questa variabile contiene la lista dei Descrittori di Task liberi. Viene gestita internamente dal Kernel e deve essere utilizzata dai Moduli di Schedulazione all'interno della Task Call `task_end` per liberare il descrittore di un task terminato con successo.

**TIME sys_tick** Se la OS Lib è inizializzata in modo *one shot* la variabile contiene il tick di sistema con cui viene programmato il real-time clock.

## 7.5 Protezione temporale

La *protezione temporale*[10] è uno degli obiettivi fondamentali raggiunti da molti algoritmi di scheduling proposti in letteratura negli ultimi anni.

La realizzazione di un Kernel Modulare pertanto non può prescindere dalla implementazione di tale importante caratteristica. In questa sezione verranno proposti tre approcci implementativi della protezione temporale all'interno di un Kernel Modulare[11] e ne saranno evidenziati vantaggi e svantaggi.

### 7.5.1 Proposta kernel-oriented

Una possibile implementazione della protezione temporale prevede che il Kernel si occupi di gestire utti gli oneri di gestione delle capacità. Questo comporta che un Modulo che intende utilizzare la protezione temporale deve indicare al Kernel delle caratteristiche con cui quantificare il tipo di protezione da effettuare sul task.

Il Kernel pertanto gestisce internamente la generazione di eventi di fine capacità che permettono di bloccare i task all'esaurirsi del proprio budget, richiamando se necessario eventuali funzioni di interfaccia che i Moduli dovranno predisporre per gestire tali eventi.

#### Vantaggi

- I Moduli non devono preoccuparsi al loro interno di gestire l'accounting delle capacità, in quanto sarà il Kernel a farlo per loro.
- Essendo il Kernel a garantire la protezione temporale, il sistema risul-

[10] Con *protezione temporale* si intende la capacità di un sistema operativo di gestire il tempo a disposizione in modo tale da evitare che le applicazioni utilizzino più tempo di quanto dichiarato al momento della propria creazione.

[11] Il terzo approccio (misto) è quello realmente implementato all'interno del Kernel Generico.

ta più protetto da eventuali errori di programmazione all'interno dei Moduli di Schedulazione.

- Il codice che implementa la protezione temporale deve essere scritto una volta sola.
- La semplificazione nella scrittura dei Moduli non è elevata, in quanto comunque i Moduli devono generare altri eventi quali deadline, riattivazioni, ecc...

**Svantaggi**

- Non tutti gli algoritmi necessitano di protezione temporale, quindi inserire la protezione temporale all'interno del Kernel Generico può appesantirne il codice.
- Esistono vari modi per specificare i parametri temporali di un task, ciascuno dei quali ha vantaggi e svantaggi. Dovendo il Kernel Generico implementarne uno, non è detto che alla fine la soluzione scelta permetta di implementare tutti i generici algoritmi di scheduling.
- L'implementazione del meccanismo di protezione comporta la modifica delle principali primitive del sistema per gestire in modo corretto l'accounting delle capacità.

### 7.5.2 Proposta module-oriented

Un'altra possibile implementazione della protezione temporale prevede che il Kernel non implementi alcun supporto per la protezione temporale. Questo approccio può essere sensato se gli algoritmi che si andranno ad implementare prevalentemente non necessitano di tale caratteristica.

Pertanto, un Modulo che intenda effettuare la gestione della capacità dovrà disporre di apposite strategie per garantire i tempi di esecuzione dei propri task.

**Vantaggi**

- Il Kernel Generico, non supportando la protezione temporale, risulta più snello e semplice da manutenere.
- Possono essere implementati approcci ibridi (quali ad esempio la condivisione delle capacità tra insiemi di task, ecc...), in modo trasparente al Kernel.
- In ogni caso l'utilizzatore dispone della protezione temporale all'interno del proprio sistema, poco importa se fornito da un Kernel o da un Modulo.

**Svantaggi**

- Il Kernel non garantisce nulla a riguardo dei tempi di calcolo dei task, pertanto i Moduli utilizzati devono essere scritti correttamente.
- Essendo l'implementazione di un Modulo indipendente da quella degli altri, se due Moduli implementano una tecnica di protezione temporale simile il codice diviene ridondante in termini sia di operazioni che di variabili locali.

### 7.5.3 Proposta mista

L'ultima proposta presentata consiste, come sempre, in una via di mezzo tra le proposte precedenti. In questo caso il Kernel Generico predispone una serie di infrastrutture per agevolare gli sviluppatori dei Moduli nell'implementazione della protezione temporale. Il Kernel Generico in ogni caso non obbliga i Moduli ad usare tali meccanismi, nel senso che i Moduli rimangono liberi di utilizzarne di propri o di non utilizzarne per nulla.

**Vantaggi**

- Chi scrive un Modulo si ritrova una possibilità di implementare in modo semplice e libero la gestione della protezione temporale.
- La ridondanza del codice viene limitata in modo notevole.
- Il Kernel Generico, supportando solo parzialmente la protezione temporale, risulta più snello del primo approccio proposto, e quindi anche ragionevolmente semplice da manutenere.

**Svantaggi**

- Come per la seconda soluzione, il Kernel non garantisce nulla a riguardo dei tempi di calcolo dei task, pertanto i Moduli utilizzati devono essere scritti correttamente.
- Il meccanismo parziale rimane presente anche quando un esplicito controllo del tempo di calcolo non è richiesto
- L'implementazione del meccanismo di protezione comporta la modifica delle principali primitive del sistema per gestire in modo corretto l'accounting delle capacità.

### 7.5.4 Implementazione della terza proposta

In questo paragrafo si forniscono alcune informazioni riguardanti l'implementazione della protezione temporale all'interno del Kernel Generico e dei Moduli di Schedulazione. L'approccio utilizzato è stato, tra quelli appena descritti, il terzo (misto).

Il Kernel Generico fornisce una gestione della capacità basata sulla generazione e la rimozione di un apposito evento temporale che impone la rischedulazione del sistema[12]. Tale evento per la precisione viene generato al momento della dispatch di un task e viene rimosso da tutte le primitive che possono comportare preemption. Ad ogni istante può esistere al più un solo evento di capacità, quello relativo al task in esecuzione sulla CPU.

Il Kernel Generico definisce per ogni task il bit `CONTROL_CAP` contenuto nel campo `control` del descrittore di task. Tale bit viene modificato dai Moduli di Schedulazione per richiedere per quel task la generazione dell'evento di capacità.

I Moduli di Schedulazione che intendono implementare la gestione delle capacità dovranno pertanto settare il bit `CONTROL_CAP` per i propri task e controllare, ogni volta che un proprio task viene deschedulato, se la capacità si è esaurita.

Le capacità vengono gestite tramite le seguenti variabili messe a disposizione dal Kernel Generico:

- Il campo `wcet` del descrittore di task contiene un valore che di solito (ma non sempre) rappresenta un tempo rappresentativo per il task collegato al descrittore[13]. Esso è presente per evitare che ogni Modulo debba mantenersi tale informazione con conseguente duplicazione delle strutture dati. Non viene utilizzato direttamente dal Kernel Generico.

- Il campo `avail_time` del descrittore di task contiene il tempo disponibile per l'esecuzione di un determinato task, ossia il budget residuo di un task. L'evento di fine capacità, che viene generato dal Kernel Generico al momento della dispatch, ha un tempo di attivazione che è uguale a `schedule_time + avail_time`. Il campo `avail_time` viene aggiornato dal Kernel Generico decrementando il valore del tempo trascorso dal momento dell'ultima dispatch del task.

[12]Il Kernel Generico non gestisce un vero e proprio evento di fine capacità, ma maschera la fine capacità con un normale epilogo dovuto ad una preemption.

[13]Ad esempio, potrebbe essere un tempo massimo di esecuzione o un tempo medio, a seconda dell'algoritmo implementato.

- La variabile `cap_timer` viene utilizzata dal Kernel Generico per sapere se un evento di capacità è o meno pendente per il task in esecuzione. Se tale variabile vale -1 non c'è nessun evento, altrimenti il valore assunto è quello dell'indice dell'evento di capacità. In ogni caso se `cap_timer` è diverso da -1 il Kernel Generico cancellerà automaticamente tale evento al momento di una rischedulazione. Il Kernel Generico non controlla *chi* ha generato l'evento puntato da `cap_timer`, quindi un Modulo di Schedulazione che non usa `CONTROL_CAP` può comunque generare un evento di capacità inserendone l'indice dentro `cap_timer`, evitando in questo modo di rimuovere esplicitamente l'evento all'interno del codice del Modulo.

- La variabile `cap_lasttime` contiene il tempo della precedente schedulazione. L'accounting dei tempi di calcolo viene effettuato sulla differenza tra `schedule_time` (tempo attuale di rischedulazione) e `cap_lasttime` (tempo precedente di rischedulazione). Come si può notare, il tempo di esecuzione delle primitive di sistema e degli eventi viene dato in account ai task che stanno eseguendo al momento.

Per quanto riguarda la funzione richiamata dall'evento di capacità, è sufficiente che essa rischeduli il sistema tramite le apposite funzioni messe a disposizione. I Moduli di Schedulazione che intendono generare autonomamente l'evento di rischedulazione e che utilizzano la variabile `cap_timer` nel modo corretto possono anche utilizzare la funzione utilizzata dal Kernel Generico. Tale funzione si chiama `capacity_timer` e non ha parametri.

### 7.5.5 Capacità negative

All'interno del Kernel Generico i valori contenuti nel campo `avail_time` possono assumere valori negativi. Questo fatto, che di per sè è teoricamente impossibile, può accadere per due motivi, legati alla gestione degli eventi della OS Lib ed alla gestione del meccanismo degli shadows.

Occorre notare che in generale, a causa delle imprecisioni dovute alla gestione degli eventi (vedi Sezione 5.7), un evento di capacità non scatterà mai all'istante *esatto* in cui la capacità termina. Essendo le letture del timer effettuate con precisione, l'effetto del ritardo all'interno della gestione degli eventi è quello di far andare il campo `avail_time` ad un valore **negativo** (in generale si può supporre che i valori negativi raggiunti siano piccoli rispetto ai tempi di calcolo in gioco).

L'altro motivo per cui le capacità possono assumere valori negativi è dovuto ad un approccio seguito nella gestione della ereditarietà delle priorità: in effetti, un task bloccante che termina la propria capacità **non** viene deschedulato se esiste un task a priorità maggiore che lo indica all'interno della propria catena degli shadow.

Questo comportamento è corretto in quanto si vuole che il task bloccante termini il più presto possibile la propria sezione critica per limitare il tempo di bloccaggio di task più importanti. Questo approccio è inoltre supportato in letteratura da risultati teorici (vedi [GB95]).

Come devono comportarsi i Moduli di Schedulazione di fronte al verificarsi di valori negativi all'interno delle capacità? Le soluzioni adottabili sono fondamentalmente due.

La prima soluzione prevede la ricarica (tramite una somma) della capacità di una quantità $Q_s$; se la capacità dopo la ricarica rimane negativa "si perde un turno". Questa soluzione è adatta a tutti gli algoritmi che prevedono un riempimento periodico (quali Round Robin, Polling Server o Deferrable Server) ma non per algoritmi tipo Sporadic Server o Constant Bandwidth Server, dove il successivo riempimento è causato da un consumo di capacità (dunque, con questi ultimi algoritmi il server non diventerebbe più attivo).

La seconda soluzione prevede comunque una ricarica al valore $Q_s$; quest'ultima soluzione è adatta a tutti i tipi di algoritmi, ma occorre tenere conto dell'incremento del fattore di utilizzazione che assume pertanto una formula simile a $\frac{Q_s+\Delta}{T_s}$.

## 7.6 Funzioni di utilità

Questa Sezione contiene la descrizione di una serie di funzioni di utilità che possono essere utilizzate per semplificare la scrittura dei Moduli. Spesso tali funzioni sono semplici ridefinizioni delle corrispondenti funzioni della OS Lib. Si consiglia di utilizzare sempre le ridefinizioni e non le funzioni originali della OS Lib all'interno dei Moduli.

### 7.6.1 Gestione degli eventi

Il Kernel Generico fondamentalmente utilizza la gestione degli eventi fornita dalla OS Lib. Le uniche differenze sono (vedi il file `include/kernel/func.h`):

- la funzione `event_post` è ridefinita nella funzione `kern_event_post`. La ridefinizione aggiunge il controllo sull'indice restituito. Nel caso non sia possibile creare un evento, una eccezione viene sollevata[14].

- viene aggiunta la funzione `event_need_reschedule`: essa non ha parametri e deve essere chiamata all'interno delle funzioni di gestione degli eventi che devono imporre una rischedulazione del sistema. I Moduli non devono mai richiamare la funzione schedule all'interno di un evento.

### 7.6.2 Gestione delle eccezioni

La gestione delle eccezioni all'interno del Kernel Generico viene realizzata basandosi sulla gestione dei segnali real-time dello standard POSIX. In particolare, le eccezioni sono implementate come la generazione di un segnale realtime di numero 9, SIGHEXC, che tipicamente scrive un messaggio a video e termina il sistema.

La funzione da utilizzare per generare una eccezione è la seguente:

[14] Il fatto che la OS Lib non abbia spazio per gestire eventi è *veramente* catastrofico. In effetti, il dimensionamento della OS Lib deve essere fatto a priori in modo da garantire sempre che la OS Lib accetti gli eventi generati.

```
void kern_raise(int n, PID p);
```

Come risultato della chiamata di tale funzione, un segnale realtime viene accodato all'interno del sistema ed il parametro `n` viene passato all'interno del campo `si_value` della struttura `siginfo_t` passata come parametro all'handler del segnale.

### 7.6.3 Gestione della Memoria

Il Kernel Generico mette a disposizione dei Moduli e delle Applicazioni un allocatore di memoria dinamica che utilizza direttamente il List-based Memory Manager (LMM) distribuito con la libreria OS Kit (Capitolo 23, [FBB+97]). Il codice della LMM si trova all'interno della directory `kernel/mem`; le definizioni delle funzioni descritte in questa Sottosezione si trovano all'interno del file `kernel/mem.c`.

Le funzioni descritte devono essere chiamate ad interruzioni disabilitate.

L'allocatore permette inoltre di distinguere tre regioni di memoria all'interno del PC; tali regioni possono essere specificate alle funzioni di allocazione tramite un valore da passare nel parametro denominato `flags` (quando disponibile nel prototipo):

- Memoria con indirizzi inferiori ad 1 Mb (valore del campo flag uguale a `MEMORY_UNDER_1M`)
- Memoria con indirizzi compresi tra 1 Mb e 16 Mb (valore del campo flag uguale a `MEMORY_FROM_1M_TO_16M`)
- Memoria con indirizzi minori di 16 Mb (valore del campo flag uguale a `MEMORY_UNDER_16M`)
- Memoria con indirizzi oltre i 16 Mb (valore del campo flag uguale a `0`)

Di seguito sono descritte le funzioni utilizzabili (in Figura 7.1 ne sono indicati i prototipi):

```
void *kern_alloc(DWORD s);
void *kern_alloc_aligned(size_t size, lmm_flags_t flags,
      int align_bits, DWORD align_ofs);
void *kern_alloc_gen(size_t size, lmm_flags_t flags,
      int align_bits, DWORD align_ofs,
      DWORD bounds_min, DWORD bounds_max);
void kern_free(void *block, size_t size);
void *kern_alloc_page(lmm_flags_t flags);
void kern_free_page(void *block);
void *DOS_alloc(DWORD size);
void DOS_free(void *ptr,DWORD size);
```

Figura 7.1: Funzioni disponibili per la gestione della memoria dinamica.

`kern_alloc` Questa funzione alloca un blocco di memoria di dimensione s byte restituendolo come `void *`. Non sono specificate particolari caratteristiche per il blocco da allocare. La funzione restituisce `NULL` se non c'è un blocco con le caratteristiche richieste.

`kern_alloc_aligned` Questa funzione alloca un blocco allineato di memoria. Il blocco ritornato è tale che i primi `align_bits` dell'indirizzo del blocco ritornato devono corrispondere agli `align_bits` più bassi di `align_ofs`. In altre parole, `align_bits` specifica un allineamento come potenza di 2, mentre `align_ofs` indica uno spiazzamento dall'allineamento "naturale". La funzione restituisce `NULL` se non c'è un blocco con le caratteristiche richieste.

`kern_alloc_gen` Questa funzione alloca un blocco allineato di memoria in modo simile a `kern_alloc_aligned`, con la condizione aggiuntiva che il blocco si debba trovare all'interno degli indirizzi che si trovano tra `bounds_min` e `bounds_max`.

`kern_free` Questa funzione libera un blocco allocato in precedenza tramite una delle funzioni di allocazione. Notare come, a differenza della funzione `free` dello standard C, la `kern_free` richieda la specifica della dimensione del blocco da liberare.

`kern_alloc_page` Questa funzione restituisce l'indirizzo di memoria di una pagina di memoria. Il blocco restituito è di 4Kb allineato ai 4Kb.

`kern_free_page` Questa funzione libera una pagina allocata in precedenza con `kern_alloc_page`.

`DOS_alloc, DOS_free` Queste funzioni possono essere utilizzate per allocare memoria al di sotto del primo Mb.

### 7.6.4 Cambi di contesto

La creazione dei contesti all'interno del Kernel Generico viene realizzata tramite funzioni analoghe a quelle della OS Lib, chiamate `kern_context_create` e `kern_context_delete`.

Per quanto riguarda i cambi di contesto, il Kernel Generico mette a disposizione due funzioni, `kern_context_load` e `kern_context_save` che svolgono i seguenti compiti:

- disabilitazione delle interruzioni;
- cambiamento di contesto vero e proprio tramite chiamata alle funzioni della OS Lib;
- dispatching dei segnali;
- test di cancellazione asincrona.

Tali funzioni non devono **mai** essere chiamate all'interno dei Moduli di Schedulazione, in quanto sono chiamate per loro dal Kernel Generico.

Quando una funzione che gestisce un evento temporrale di un Moduli di Schedulazione ha bisogno di rischedulare il sistema deve essere richiamata la apposita funzione `event_need_reschedule`.

Le funzioni descritte in questo paragrafo si trovano all'interno del file `include/kernel/func.h`.

### 7.6.5 Code, array e puntatori

Per semplificare la gestione delle code e dei riferimenti interni al kernel si è seguito il seguente approccio:

- I descrittori quando possibile sono stati allocati in modo statico.
- Gli identificatori che rappresentano elementi di array statici sono stati indicati con interi (si pensi ai tipi PID, LEVEL, RLEVEL, per citare i più usati).
- Dove possibile si è evitato l'uso di strutture dati allocate dinamicamente.

Per gestire il tipo di code più comuni, le code di task, si è deciso di fornire una serie di funzioni che agevolino l'accodamento dei task risparmiando all'utente la scrittura delle apposite funzioni. Tali funzioni presuppongono l'utilizzo dei campi `next` e `prev` del descrittore di task, nonchè dei campi `timespec_priority` e `priority`.

Le code sono implementate con un doppio link, in modo tale da rendere il tempo di estrazione costante al variare del numero di elementi in coda.

I prototipi delle funzioni che gestiscono tali code sono indicati in Figura 7.2. Essi sono contenuti nei files `kernel/queue.c`, `kernel/qqueue.c`, `include/kernel/func.h`, descritti di seguito. I tipi di dato utilizzati per gestire le code di task sono due:

`QUEUE` rappresenta una coda di task di cui viene mantenuta traccia solo del primo elemento. Utile per implementare code prioritarie.

`QQUEUE` rappresenta una coda di task di cui viene mantenuta traccia sia del primo che dell'ultimo elemento. Utile per implementare code in cui

```
/* Simple QUEUE management functions */
void q_insert           (PID p, QUEUE *q);
void q_timespec_insert  (PID p, QUEUE *q);
void q_extract          (PID p, QUEUE *q);
PID  q_getfirst         (       QUEUE *q);
void q_insertfirst      (PID p, QUEUE *q);

/* QQUEUE management functions */
void qq_init            (       QQUEUE *q);
void qq_insert          (PID p, QQUEUE *q);
void qq_timespec_insert (PID p, QQUEUE *q);
void qq_extract         (PID p, QQUEUE *q);
PID  qq_getfirst        (       QQUEUE *q);
void qq_insertfirst     (PID p, QQUEUE *q);
void qq_insertlast      (PID p, QQUEUE *q);
PID  qq_queryfirst      (       QQUEUE *q);
PID  qq_querylast       (       QQUEUE *q);
```

Figura 7.2: Prototipi delle funzioni di gestione delle code.

si inserisce anche in fondo alla coda (ad esempio quelle di un classico scheduler Round Robin).

Le variabili di tipo `QUEUE` vengono inizializzate tipicamente a `NIL`. Le variabili di tipo `QQUEUE` vengono inizializzate tramite la funzione `qq_init`.

Il significato delle funzioni è il seguente:

`q_insert, qq_insert` Inseriscono il task `p` all'interno della coda, mantenendo la coda ordinata in base al campo `priority` del descrittore di task.

`q_timespec_insert, qq_timespec_insert` Inseriscono il task `p` all'interno della coda, mantenendo la coda ordinata in base al campo del descrittore di task `timespec_priority`.

`q_extract, qq_extract` Estrazione dell'elemento `p` dalla coda; si suppone che l'elemento `p` sia *realmente* dentro la coda.

`q_getfirst, qq_getfirst` Estrazione del primo elemento in coda.

`q_insertfirst, qq_insertfirst` Inserimento di un elemento in prima posizione.

`qq_insertlast` Inserimento di un task in fondo alla coda.

`qq_queryfirst, qq_querylast` Lettura del primo e dell'ultimo elemento in coda

### 7.6.6 Funzioni di inizializzazione

Il Kernel Generico permette di specificare delle funzioni da richiamare in momenti cruciali della vita del sistema, quali ad esempio l'inizializzazione e la terminazione del sistema (vedi Sezione 10.2). Tali funzioni possono essere registrate tramite la seguente primitiva di sistema:

```
int sys_atrunlevel(void (*f)(void *),void *p, BYTE when);
```

I parametri specificabili per questa funzione sono:

`f` Indica la funzione da registrare;

`p` Indica il parametro da passare alla funzione `f` al momento della sua esecuzione;

`when` Indica il momento in cui tale funzione deve essere richiamata. I valori possibili sono i seguenti:

- `RUNLEVEL_INIT` La funzione viene richiamata non appena il Kernel passa in modalità multitasking;
- `RUNLEVEL_SHUTDOWN` La funzione viene richiamata in seguito alla esecuzione della funzione `sys_abort` o `sys_end`; Il sistema è ancora in modalità multitasking;
- `RUNLEVEL_BEFORE_EXIT` La funzione viene richiamata subito prima dell'uscita del Kernel dalla modalità multitasking;
- `RUNLEVEL_AFTER_EXIT` La funzione viene richiamata alla fine della vita del sistema, subito prima di uscire dal Kernel per ritornare al sistema operativo ospite (se si è utilizzato l'extender proprietario per far partire l'applicazione) o far bloccare il sistema (se si è utilizzato un boot loader a standard multiboot).

È possibile infine specificare con un `OR` bit a bit sul campo `when` il flag `NO_AT_ABORT` che disabilita la chiamata delle funzioni di registrazione nel caso si stia uscendo dal sistema tramite la funzione `sys_abort`.

Possono essere registrate al massimo `MAX_RUNLEVEL_FUNC` funzioni.

### 7.6.7 Disabilitazione delle interruzioni

In questa sezione sono indicate una serie di funzioni di utilità utilizzate nei Moduli e nel Kernel Generico.

Per prima cosa, il Kernel ridefinisce tramite semplici macro (nel file `include/kernel/func.h`) le seguenti funzioni della OS Lib:

`cli` rinominata in `kern_cli`

`sti` rinominata in `kern_sti`

`ll_fsave` rinominata in `kern_fsave`

`ll_frestore` rinominata in `kern_frestore`.

Tali ridefinizioni risultano utili per effettuare il debug in situazioni critiche, quindi è sempre preferibile utilizzarle all'interno del proprio codice.

Per visualizzare a video dei messaggi il kernel Generico mette a disposizione le due funzioni `printk` e `kern_printf`. Tali funzioni hanno un formato simile alla funzione `printf` dello standard C con l'unica differenza che la stampa viene effettuata direttamente a video. La funzione `printk` permette anche di specificare l'importanza del messaggio da stampare in funzione di alcune costanti specificabili all'inizio del primo parametro. Per ulteriori informazioni fare riferimento ai file `include/kernel/func.h` e `include/kernel/log.h`.

Infine, la terminazione del sistema in modo "pulito" può essere ottenuta tramite le primitive `sys_end` e `sys_abort`. Per ulteriori informazioni fare riferimento al file `include/kernel/func.h` ed alla Sezione 10.2.

# Chapter 8

# I moduli di schedulazione

In questo capitolo viene descritta nel dettaglio l'interfaccia di un generico Modulo di Schedulazione. In particolare viene descritta la semantica delle funzioni che compongono l'interfaccia e vengono fornite una serie di convenzioni utilizzate nella scrittura di tali moduli. Per le informazioni architetturali riguardanti i Moduli di Schedulaizone fare riferimento alle Sezioni 4.2.

## 8.1 Ciclo di vita di un task

Per comprendere fino in fondo il significato dell'interfaccia di un Modulo di Schedulazione occorre per prima cosa puntualizzare quali sono gli eventi che caratterizzano in generale la vita di un task, dalla sua creazione alla sua morte. Di fatto l'interfaccia dei moduli rispecchia molto da vicino la successione di tali eventi[1]. Di seguito verranno descritti gli eventi più importanti utilizzando figure esemplificative.

La Figura 8.1 mostra un primo scenario "di base". Viene infatti considerato un task che, una volta creato ed attivato, subisce preemption da parte di un altro task che muore poco dopo la sua prima attivazione. Gli eventi

[1] Le corrispondenti funzioni sono chiamate in modo identico all'evento, con in più il prologo `task_`.

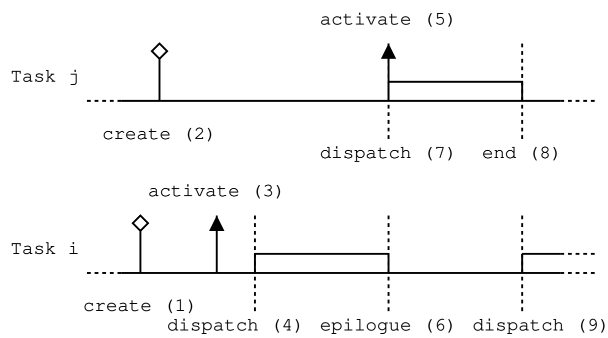


Figure 8.1: Scenario di base: i task i e j vengono creati ed attivati. Il task i viene messo in esecuzione e, poco dopo, subisce preemption da parte del task j. I numeri tra parentesi indicano la successione degli eventi.

coinvolti sono i seguenti:

**create** Tale evento viene generato alla creazione del task. Il Modulo di Schedulazione predispone le strutture dati in modo tale che tutto sia pronto per la successiva attivazione del nuovo nato.

**activate** Tale evento viene generato in seguito alla prima attivazione esplicita del task all'interno del codice utente ed autorizza il Modulo di Schedulazione a schedulare tale task per l'esecuzione.

**dispatch** Questo evento viene generato quando il task, una volta schedulato[2], viene scelto dal Kernel Generico per essere messo in esecuzione[3]. Il Modulo di Schedulazione predispone quindi le proprie strutture dati per far si che il task possa essere messo in esecuzione (ad esempio, estrae il task dalla coda pronti, se c'è).

**epilogue** Questo evento viene generato tutte le volte che il task subisce preemption da parte di un altro task. Il Modulo di Schedulazione deve

[2]Gli eventi di schedulazione, come altri eventi, non sono presentati in figura.

[3]Non è detto che il task schedulato sia quello su cui viene fatta la dispatch (cfr. Sezione 4.4.5).

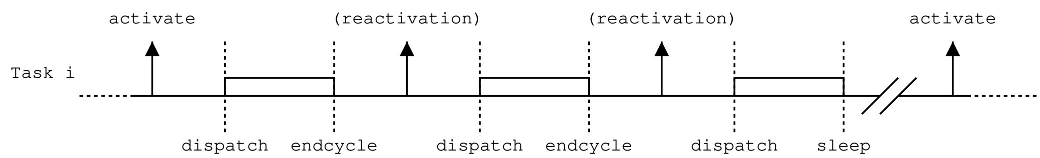


Figure 8.2: Gestione delle attivazioni. In figura il task (periodico) viene attivato una volta ed esegue per tre attivazioni prima di fermarsi con un evento di sleep. Per essere rieseguito il task necessita di una poi di una attivazione esplicita.

pertanto rimuovere dalla esecuzione il task in oggetto e rimetterlo nello stato ready.

**end** Tale evento viene generato alla morte naturale del task, cioè quando in generale il task arriva alla fine del suo corpo. Il Modulo di Schedulazione è autorizzato ad eliminare ogni traccia del task dalle proprie strutture dati.

Notare come lo standard POSIX non preveda la distinzione tra creazione ed attivazione di un task, in quanto la primitiva messa a disposizione (la `pthread_create`) crea ed attiva direttamente il task.

In Figura8.2 vengono invece considerati gli eventi riguardanti la gestione delle attivazioni dei task. Lo scenario considerato è quello di un task che, dopo essere stato attivato, esegue per tre attivazioni consecutive prima di addormentarsi in attesa di un'altra attivazione esplicita. I nuovi eventi introdotti in questo scenario sono i seguenti:

**endcycle** L'evento endcycle corrisponde ad una terminazione della attivazione corrente, segnalata al Kernel tramite una apposita primitiva inserita all'interno del codice. Se il task ha una semantica di tipo periodico questo evento implica anche che il task verrà riattivato a cura del Modulo di Schedulazione all'ingresso del periodo successivo. Se il task è invece di tipo aperiodico questo evento implica che il task si poga in attesa di una attivazione esplicita da parte dell'utente (tale attivazione esplicita viene comunicata al modulo tramite un evento activate).

**sleep** L'evento sleep corrisponde ad una terminazione della attivazione corrente. In ogni caso il task si pone in attesa di una attivazione esplicita da parte dell'utente, segnalata al Kernel tramite una apposita primitiva inserita all'interno del codice. Eventuali riattivazioni interne programmate nell'immediato futuro da parte del Modulo di Schedulazione devono essere annullate.

**(reactivation)** Questo evento non viene generato direttamente dall'utente tramite l'inserimento di una apposita primitiva nel codice ma viene gestito internamente al Modulo di Schedulazione che gestisce il task. I compiti di questo evento sono quelli di riattivare il task al termine del periodo in cui è stata gestita con successo una attivazione.

A prima vista gli eventi *sleep* ed *endcycle* sembrano identici. In effetti esiste una sottile differenza nel caso in cui sia implementata una semantica che prevede il salvataggio delle attivazioni pendenti.

Con attivazione pendente si indica un evento *activate* generato quando ancora non è terminata la precedente attivazione del task. La gestione delle attivazioni pendenti è lasciata all'implementatore del Modulo di Schedulazione. Nel caso in cui le attivazioni vengano salvate, gli eventi *endcycle* e *sleep* (che indicano la terminazione di una attivazione) differiscono:

- In presenza di attivazioni pendenti l'evento *endcycle* non comporta per il task l'attesa dell'inizio del periodo successivo. In pratica il significato di una attivazione pendente è simile alla perdita di un periodo, che in questo modo viene "recuperato".

- In presenza di attivazioni pendenti l'evento *sleep* comporta la sospensione del task e la cancellazioni di eventuali attivazioni pendenti, che pertanto non verranno gestite.

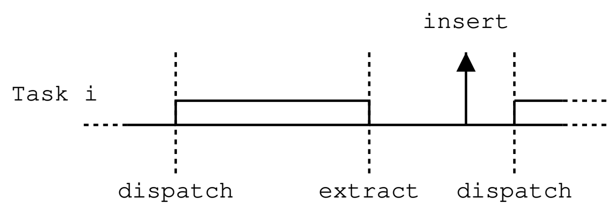


Figure 8.3: Sincronizzazione tra task. In figura il task effettua una chiamata ad una primitiva di sincronizzazione, che comporta il suo momentaneo bloccaggio tramite l'evento extract. Quando effettivamente il task potrà continuare l'esecuzione, un altro evento di tipo insert verrà generato.

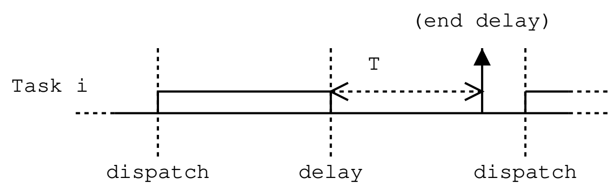


Figure 8.4: Sospensione del task per un determinato intervallo di tempo T.

In Figura 8.3 vengono considerati gli eventi di sincronizzazione. I nuovi eventi considerati sono gli eventi *extract* ed *insert*, descritti di seguito:

**extract** L'effetto di questo evento è quello di disabilitare momentaneamente la schedulabilità di un task.

**insert** Questo evento indica che il punto di sincronizzazione è stato raggiunto e pertanto è necessario riabilitare la schedulabilità del task.

Il Kernel Generico assicura che non sarà mai richiamato un evento *insert* senza che prima non sia stato generato sul task un evento *extract*.

Infine, il Kernel Generico fornisce ad un task la possibilità di autosospendersi in modo non interrompibile per un determinato intervallo minimo di tempo[4]. In Figura 8.4 e di seguito sono evidenziati e descritti gli eventi interessati.

[4]La presenza di questa primitiva è derivata da una necessità di compatibilità con le vecchie versioni del Kernel Hartik; in realtà essa può essere sostituita anche se con alcune differenze, dalla primitiva `nanosleep` dello standard POSIX.

**delay** Questo evento viene generato in seguito all'inserimento di una apposita primitiva all'interno del Kernel, ed indica che il task in oggetto intende disabilitare la propria schedulazione per un certo intervallo di tempo.

**(end delay)** Questo è un evento generato internamente dai Moduli di Schedulazione per riabilitare il task alla fine della pausa.

## 8.2 Convenzioni sulle code di task

I moduli scritti e distribuiti con il Kernel seguono le seguenti convenzioni riguardanti le code di task:

- Le variabili di tipo `QUEUE` e `QQUEUE` che contengono i primi elementi in coda sono allocate all'interno dell'estensione del descrittore di livello;
- Il task in esecuzione non è contenuto in alcuna coda; ovvero, un task viene estratto dalla coda pronti (se ne esiste una all'interno del Modulo) al momento del suo dispatch;
- I Moduli che implementano gli Algoritmi di Schedulazione utilizzano le funzioni messe a disposizione dal Kernel Generico per la gestione delle code, pertanto i moduli che implementano i Server Aperiodici non possono modificare i campi `next` e `prev` del descrittore di Task quando un loro task è inserito in un altro Modulo;
- Un modulo può gestire code ed altre strutture dati diverse da quelle messe a disposizione dal Kernel Generico. In questo caso le strutture dati in questione dovranno essere inserite all'interno dell'estensione del descrittore del livello e le funzioni per manipolarle dovranno essere visibili solo all'interno del Modulo.

## 8.3 Funzioni di interfaccia di un Modulo di Schedulazione

Di seguito sono descritte le funzioni che caratterizzano l'interfaccia di un Modulo di Schedulazione. Per ogni funzione saranno fornite indicazioni generali riguardanti il suo utilizzo, oltre ad esempi riguardanti possibili implementazioni. Per l'organizzazione dell'interfaccia dei Moduli di Schedulazione fare riferimento alle Sezioni 4.2.4 e 7.2.2.

La convenzione seguita nell'assegnazione dei nomi delle funzioni è stata quella di far precedere ogni nome da un prefisso che indichi il tipo di funzione (`level_` per le Level Calls, `task_` per le Task Calls, `guest_` per le Guest Calls). Come anticipato nella Sezione 7.2.2, tutte le funzioni di interfaccia ricevono come primo parametro una variabile di tipo `LEVEL`, utilizzata per risalire al descrittore del Modulo di Schedulazione; eventualmente possiedono inoltre alcuni parametri aggiuntivi.

Le funzioni indicate in Tabella 8.1 devono supporre che il tempo attuale si trovi all'interno della variabile `schedule_time`, evitando quindi di leggere direttamente il tempo con la apposita funzione `ll_gettime` messa a disposizione dalla OS Lib.

La Level Call , le task Calls e le Guest Call ,

Tutte le funzioni dei Moduli di Schedulazione eccetto la `sys_status` sono richiamate ad **interruzioni disabilitate**.

### 8.3.1 Level Calls

```
int level_accept_task_model(LEVEL l, TASK_MODEL *m);
```

Questa funzione riceve come parametro un puntatore ad un `TASK_MODEL`.

La funzione è richiamata dalla primitiva `task_create` in quanto al momento della creazione occorre ricercare un Modulo di Schedulazione capace di gestire un determinato Modello di Task.

| Tipo | Funzione |
|---|---|
| Level Call | `level_scheduler` |
| Task Call | `task_dispatch`<br>`task_epilogue`<br>`task_endcycle`<br>`task_sleep`<br>`task_extract`<br>`task_delay` |
| Guest Call | `guest_dispatch`<br>`guest_epilogue`<br>`guest_endcycle`<br>`guest_sleep`<br>`guest_extract`<br>`guest_delay` |

Table 8.1: Funzioni che leggono il tempo utilizzando la variabile `schedule_time`.

La funzione restituisce un valore `0` se il Modulo di Schedulazione al livello e può gestire il modello m, `-1` altrimenti.

Si ricorda che un Modulo di Schedulazione può accettare anche più di un Modello di Task. Il reale Modello passato nel puntatore può essere dedotto dal valore del campo `pclass` della struttura `TASK_MODEL`. Tipicamente se un Modulo di Schedulazione accetta una certa `pclass`, allora lo stesso Modulo deve accettare il valore `pclass|l` (cfr. Sezione 6.2.1).

```
int level_accept_guest_model(LEVEL l, TASK_MODEL *m);
```

Questa funzione riceve come parametro un puntatore ad un `TASK_MODEL`.

La funzione può essere richiamata da un generico Modulo di Schedulazione[5] per sapere se un determinato Modello di Task possa o meno essere accettato.

La funzione restituisce un valore `0` se il Modulo di Schedulazione al livello

[5]che intende inserire un task come ospite (guest) all'interno del Modulo corrente.

l può gestire il modello m per i guest task, `-1` altrimenti.

Si ricorda che un Modulo di Schedulazione può accettare anche più di un Modello di Task. Il reale Modello passato nel puntatore può essere dedotto dal valore del campo `pclass` della struttura `TASK_MODEL`. Tipicamente se un Modulo di Schedulazione accetta una certa `pclass`, allora lo stesso Modulo deve accettare il valore `pclass|l` (cfr. Sezione 6.2.1).

```
void level_status(LEVEL l);
```

Questa funzione non ha parametri aggiuntivi e tipicamente realizza la stampa a video di informazioni statistiche riguardanti il livello l. Ad esempio, un tipico output di questa funzione prevede l'indicazione di parametri quali la banda utilizzata dal Modulo, alcuni flag, i task gestiti dal Modulo con eventuali informazioni aggiuntive (periodo, deadline, attivazioni pendenti, ecc...).

Questa funzione è stata direttamente ereditata dalle precedenti versioni del Kernel Hartik, e probabilmente subirà alcune modifiche nelle versioni future del Kernel.

```
PID level_scheduler(LEVEL l);
```

Questa funzione non ha parametri aggiuntivi ed implementa lo schedulatore del Modulo.

Lo scopo di questa funzione è pertanto quello di indicare il prossimo task da schedulare tra quelli gestiti all'interno del Modulo[6]. Se il livello non implementa uno schedulatore (ad esempio perchè i propri task vengono sempre inseriti in altri Moduli) o non possiede al momento task da schedulare, il valore ritornato deve essere `NIL`.

La scelta riguardante il task da schedulare deve essere realizzata utilizzando le sole informazioni interne al Modulo, *prescindendo da altri Moduli presenti nel sistema.* Il Fatto che uno schedulatore di livello indichi un task

[6]Si ricorda che un Modulo di Schedulazione gestisce la schedulazione sia dei propri task che dei task inseriti da altri Moduli mediante le guest calls.

da schedulare non implica che tale task venga messo in esecuzione subito dopo (vedi Sezione 4.4.5). Pertanto lo schedulatore di livello non deve:

- Modificare il puntatore del task in esecuzione (contenute nelle variabili `exec` ed `exec_shadow`);
- Gestire i timer riguardanti deadline, capacità, ecc.;
- Predisporre le strutture dati per l'esecuzione del task indicato (ad esempio, se è presente nel Modulo una coda pronti, il task indicato non deve essere tolto da tale coda).

```
int level_guarantee(LEVEL l, bandwidth_t *freebandwidth);
```

Questa funzione implementa la garanzia on-line per i task *posseduti* dal Modulo[7]. La funzione riceve un parametro aggiuntivo che rappresenta la banda attualmente allocata nel sistema dai Moduli di Schedulazione con livello minore di `l` (ossia con priorità più elevata).

Le procedure di garanzia implementabili appartengono alla tipologia che utilizza come parametro fondamentale il fattore di utilizzazione. Eventuali altri approcci devono essere supportati tramite garanzia off-line.

La funzione restituisce 1 se i task del Modulo e quelli appartenenti a Moduli di livello minore sono garantiti; ritorna 0 altrimenti. Il parametro freebandwidth deve essere aggiornato dalla funzione per comprendere la banda utilizzata dai task appartenenti al Modulo.

La funzione può essere registrata all'interno del Modulo con valore NULL per indicare che il Modulo corrente ed i successivi non necessitano l'esecuzione di procedure di garanzia (vedi Sezione 10.3).

Questa funzione viene richiamata dal Kernel Generico per effettuare una garanzia ogni volta che un task è creato all'interno del sistema. La chiamata viene effettuata dopo aver creato il task tramite la task call `task_create`.

[7]La procedura di garanzia non deve riguardare i task inseriti nel Modulo corrente tramite le guest calls.

Le funzioni di garanzia vengono richiamate in cascata indipendentemente dal Modulo di appartenenza del nuovo task (vedi Sezione 10.3).

### 8.3.2 Task Calls

Di seguito sono elencate le Task Call presenti nel sistema. Occorre far notare due aspetti fondamentali:

- Le funzioni `task_epilogue`, `task_endcycle`, `task_sleep`, `task_end`, `task_extract`, `task_delay` sono chiamate alla fine di ogni slice temporale in cui un task è eseguito. Per ogni slice viene riciamata solo una di tali funzioni, cioè esse sono in alternativa;

- Un Modulo di Schedulazione per semplicità può decidere di non implementare alcune funzioni dell'interfaccia. Occorrerà però avvertire adeguatamente l'utente del modulo di questo fatto, in modo tale che non inserisca primitive che generano gli eventi ad esse collegate. Tipicamente una funzione non implementata si limita a generare una eccezione di tipo `XUNVALID_TASK`, in modo tale da poter accorgersi dell'accaduto (in ogni caso non è consigliabile impostare il puntatore a funzione nel descrittore ad un valore `NULL`). Le funzioni "eliminabili" sono elencate di seguito: `task_sleep`, `task_endcycle`, `task_delay`, `task_extract`, `task_insert`.

```
int task_create(LEVEL l, PID p, TASK_MODEL *m);
```

Questa funzione viene richiamata dal Kernel Generico all'interno della primitiva `task_create` per creare un task all'interno di un Modulo. La funzione accetta come parametri l'indice del descrittore allocato dal Kernel per il nuovo task ed il Modello di Task passato tramite la primitiva di creazione.

Il Kernel Generico garantisce che il Modello di Task passato in questa funzione appartiene ai Modelli di task gestibili dal Modulo[8]. Il valore ritornato dalla funzione è `0` se il task può essere creato all'interno del livello, `-1` se il task non può essere creato.

La funzione ha il compito di inizializzare le strutture dati locali al Modulo con i parametri di Qualità del Servizio passati all'interno del Modello di Task. La funzione **non deve** abilitare il task ad essere schedulato all'interno del sistema (se ad esempio è presente una coda di task pronti, il nuovo task non dovrebbe essere inserito in tale coda).

La primitiva `task_create` del Kernel Generico imposta di default lo stato del nuovo task su `SLEEP`, e setta i flag del campo `control` al valore impostato nel campo `control` del Modello di Task `TASK_MODEL`. La funzione può eventualmente modificare come crede tali impostazioni.

La possibilità di creare un task all'interno di un Modulo non entra nel merito della garanzia del sistema. In effetti la procedura di garanzia viene richiamata subito dopo questa funzione, pertanto se un nuovo task non può essere garantito la notifica di fallita creazione potrà essere fornita a scelta da questa funzione o dalla funzione `level_guarantee`. Il motivo di fallimento della task_create è tipicamente la mancanza di memoria dinamica disponibile.

```
void task_detach(LEVEL l, PID p);
```

Questa funzione è richiamata all'interno della primitiva del Kernel Generico `task_create` nel caso in cui si sia verificato un errore all'interno della procedura di creazione del nuovo task.

La funzione riceve come parametro l'indice del descrittore di task su cui poco prima era stata chiamata con successo la task call `task_create` . Essa ha il compito di cancellare ogni traccia del nuovo task all'interno del sistema, oltre a liberare eventuali blocchi di memoria allocati durante la procedura di

[8]In effetti, il Kernel Generico chiama la funzione dopo aver verificato tramite la `level_accept_task_model` che il Modulo possa accettare il nuovo task

creazione.

```
int task_eligible(LEVEL l, PID p);
```

Questa funzione viene richiamata dal Kernel Generico all'interno della procedura di scheduling e serve per assicurare il Kernel della correttezza del valore ritornato dallo scheduler di livello.

La funzione riceve come parametro un indicatore del task appena ritornato dall'ultima chiamata ad uno schedulatore di livello. Essa restituisce 0 se il task può essere schedulato, -1 altrimenti. Restituire un valore -1 implica che il sistema richiama nuovamente lo scheduler di livello per scegliere un nuovo task.

Tipicamente questa funzione è utilizzata nella implementazione dei server aperiodici quando si ha necessità di conoscere se un proprio task è stato scelto dal level_scheduler del Modulo Master, per eventualmente correggerne parametri non più aggiornati.

Occorre notare come tale funzione spesso non è necessaria in configurazioni del sistema per cui la teoria garantisce la schedulabilità dell'intero sistema. Essa risulta utile nello studio di casi "patologici" o di situazioni impreviste. Ad esempio (vedi Figura 8.5), supponiamo di implementare in un Modulo un algoritmo tipo CBS in modo tale che si appoggi ad un altro Modulo di Schedulazione. Il Modulo CBS inserirà i propri task mediante le guest calls, dopo di che aspetterà che i propri task vengano schedulati. In generale può succedere che a causa di elevato carico il task schedulato con CBS venga per qualche strana ragione schedulato ad un tempo successivo alla propria deadline. A questo punto il Modulo CBS può decidere tramite la funzione `task_eligible` di posticipare la deadline e richiedere la rischedulazione del sistema per tenere conto delle modifiche effettuate.

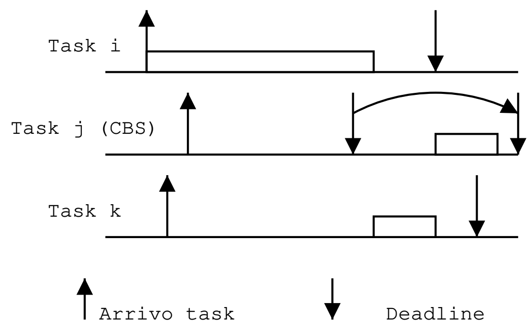


Figure 8.5: Utilizzo della funzione `task_eligible`. Supponiamo di aver scritto un Modulo di Schedulazione che implementi un algoritmo CBS appoggiandosi ad un Modulo EDF preesistente. Supponiamo di voler utilizzare il Modulo CBS *senza modifiche* con un nuovo Modulo EDF-NP (non preemptive) per il quale non è stata implementata alcuna procedura di garanzia. Il Modulo EDF-NP schedulerà inizialmente il task i. Terminata l'istanza di tale task, il task a deadline minore diventa quello j (CBS), che però adesso ha una deadline "obsoleta". La funzione `task_eligible` chiamata sul Modulo CBS quando il modulo EDF-NP cerca di schedulare il task j permette al Modulo CBS di effettuare un posticipo della deadline permettendo al task k di eseguire correttamente.

```
void task_dispatch(LEVEL l, PID p, int nostop);
```

Questa funzione è richiamata dal Kernel Generico per indicare ad un Modulo che un determinato task `p` sta per essere messo in esecuzione.

Quando questa funzione viene richiamata non è più possibile far si che il task non venga eseguito. Se si vuole intervenire in modo da evitare la schedulazione di un task occorre agire a livello della funzione `task_eligible`. Non è possibile impedire l'esecuzione di un task che si trova alla fine di una catena di shadow (vedi Sezione 4.4.5).

La funzione riceve due parametri: il task da mettere in esecuzione subito dopo ed un parametro `nostop` che vale `0` se `exec == exec_shadow`, un valore diverso altrimenti.

In pratica la funzione `task_dispatch` rappresenta un vero e proprio prologo con cui i Moduli di Schedulazione predispongono le strutture dati interne per far si che il task possa andare in esecuzione correttamente. Lo stato del task `p` viene settato ad `EXE` prima della chiamata della funzione. La funzione non deve modificare i valori delle variabili `exec`, `exec_shadow`, e lo stato del task.

Di seguito sono elencate azioni tipiche che vengono compiute in questa funzione:

- Tipicamente il task è rimosso dalla coda pronti, se ne esiste una;
- Se il Modulo non utilizza il flag `CONTROL_CAP` (vedi 7.5) può essere necessario creare con la funzione `kern_event_post` un evento di fine capacità. Occorre ricordarsi che in generale tale evento non va creato se il parametro `nostop` è diverso da `0`[9];
- Se il Modulo di Schedulazione è "accoppiato" con un Modulo di Gestione delle risorse dovranno in questa funzione essere gestiti eventuali

[9] In questo caso il task sta eseguendo in quanto detiene una risorsa necessaria ad un task più importante, e quindi il task dovrà eseguire senza interruzione fino al rilascio di tale risorsa.

cambiamenti di ceiling (ad esempio questo succede in protocolli come SRP).

```
void task_epilogue(LEVEL l, PID p);
```

Questa funzione viene richiamata dal Kernel Generico nei seguenti casi:

- Il task attualmente in esecuzione ha subito preemption da parte di un altro task del sistema;
- È stata chiamata una primitiva che potenzialmente può provocare una preemption (pertanto viene richiamato lo scheduler che come prima cosa chiama questa funzione);
- È terminata la capacità del task in esecuzione (l'esaurimento della capacità appare come una richiesta di preemption all'interno del sistema.

In generale questa funzione riceve come parametro aggiuntivo il nome del task attualmente in esecuzione. L'effetto della chiamata può eventualmente disabilitare la schedulabilità del task. Al momento della chiamata della funzione il task p si trova nello stato EXE.

Di seguito sono elencate alcune azioni tipiche che vengono compiute in questa funzione:

- Se il task `p` può venire rischedulato esso viene reinserito all'interno della coda pronti, se ne esiste una all'interno del Modulo;
- Lo stato del task `p` viene di solito modificato per portarlo in uno stato interno (ad esempio uno stato interno di `READY`);
- Viene controllato se il task eventualmente ha terminato la capacità: se la capacità è terminata devono essere effettuate le opportune operazioni, quali ad esempio sollevare una eccezione, posticipare una deadline, ecc.;

- Se il Modulo ha creato un proprio evento di fine capacità senza utilizzare il flag `CONTROL_CAP` può essere necessario rimuovere l'evento di capacità creato.

```
void task_activate(LEVEL l, PID p);
```

Questa funzione viene richiamata dal Kernel Generico in seguito ad una attivazione esplicita di un task effettuata chiamando la primitiva `task_activate` o la primitiva `group_activate`.

La funzione riceve un parametro aggiuntivo p che indica il task da attivare. La funzione ha come effetto quello di rendere attiva la schedulabilità di un task.

Le attività svolte dalla funzione sono le seguenti:

- Innanzitutto viene fatto un controllo per verificare che il task sia effettivamente in uno stato compatibile con la attivazione (ad esempio, attivare un task sporadico che ha terminato una attivazione prima del proprio periodo può generare una eccezione);

- Lo stato del task, che di solito si trova in `SLEEP`, viene modificato (ad esempio diventa uno stato interno di `READY`);

- La funzione deve aggiornare il campo `request_time` del descrittore di task leggendo il tempo di sistema con la funzione `ll_gettime`;

- Il task è inserito in coda pronti (se il Modulo ne ha una);

- Se il Modulo gestisce il concetto di attivazione pendente ed il task non ha ancora finito quella corrente, l'attivazione dovrebbe essere conteggiata per il task;

- Se il Modulo deve gestire task periodici e/o con deadline, devono essere creati qui degli eventi per gestire le riattivazioni o il superamento delle deadline.

```
void task_extract(LEVEL l, PID p);
```

La funzione implementa il bloccaggio del task `p` (in esecuzione) su di una generica primitiva di sincronizzazione non implementata con il meccanismo degli shadow. La funzione accetta un parametro aggiuntivo che altro non è che il task che si sta bloccando.

L'effetto di questa funzione deve essere quello di disabilitare la schedulabilità del task fino al momento in cui esso non sarà "liberato" tramite una chiamata alla funzione `task_insert`.

Di seguito sono elencate le azioni tipiche che vengono compiute in questa funzione:

- Il task deve essere rimosso dalla coda pronti se esso non è stato rimosso dalla `task_dispatch`;
- Devono essere rimossi gli eventuali eventi creati al momento della dispatch (ad esempio gli eventi di capacità, ma non quelli di deadline o riattivazione);
- La funzione non deve modificare lo stato del task `p`, in generale ciò è fatto dalla funzione che chiama la `task_extract` stessa;

```
void task_insert(LEVEL l, PID p);
```

L'effetto della chiamata di questa funzione è duale rispetto alla funzione `task_extract`.

Tale funzione accetta un parametro aggiuntivo `p` che indica che tale task ha terminato la sincronizzazione iniziata con la chiamata della `task_extract`. In seguito alla chiamata di questa funzione il task può di nuovo venire schedulato all'interno del sistema.

La funzione si preoccupa tipicamente di inserire il task all'interno delle code interne (se presenti), modificando inoltre lo stato del task (ad esempio portandolo nello stato `READY`). Di solito la funzione non comporta la creazione di eventi.

La funzione è richiamata all'interno delle primitive del kernel che implementano sincronizzazioni senza utilizzare i meccanismi degli shadow.

```
void task_endcycle(LEVEL l, PID p);
```

Questa funzione implementa la fine di una attivazione del task `p` passato come parametro aggiuntivo.

Se il task non ha attivazioni pendenti (o se il Modulo non le gestisce) l'effetto della funzione è quello di disabilitare la schedulabilità del task in attesa di una riattivazione esplicita da parte dell'utente[10] oppure di una riattivazione automatica da parte del Modulo (nel caso di task periodici).

Se il task ha delle attivazioni pendenti la funzione deve riattivare il task comportandosi in modo molto simile alla task call `task_epilogue`, cioè facendo in modo che il task rimanga ancora schedulabile.

Di seguito sono elencate le azioni tipiche che vengono compiute da questa funzione:

- Lo stato del task viene modificato per portarlo, nel caso in cui la schedulazione debba essere disattivata, in stati "di parcheggio" (quali, ad esempio, `IDLE` o `SLEEP`);
- Il task viene rimosso da eventiuali code pronti (se il task vi si trova ancora inserito);
- Generalmente vengono gestite le capacità residue (ad esempio per implementare procedure di resource reclaiming);
- Vengono inoltre gestiti i timer di fine capacità (se creati senza utilizzare il flag `CONTROL_CAP`) e quelli sulla deadline se presenti.

[10] con una primitiva che comporti la chiamata della task call `task_activate`.

```
void task_sleep(LEVEL l, PID p);
```

Questa funzione implementa la fine di una attivazione del task `p` passato come parametro aggiuntivo.

Se il task non ha attivazioni pendenti (o se il Modulo non le gestisce) l'effetto della funzione è quello di disabilitare la schedulabilità del task in attesa di una riattivazione esplicita da parte dell'utente. Eventuali riattivazioni automatiche da parte del Modulo (ad esempio nel caso di task periodici) devono essere disabilitate. Se il task ha invece delle attivazioni pendenti tali attivazioni devono essere azzerate.

Di seguito sono elencate le azioni tipiche che vengono compiute da questa funzione:

- Lo stato del task viene modificato per portarlo, nel caso in cui la schedulazione debba essere disattivata, in stati "di parcheggio" (quale, ad esempio, `WAIT` o `SLEEP`);
- Generalmente vengono gestite le capacità residue (ad esempio per implementare procedure di resource reclaiming);
- Il task viene rimosso da eventiuali code pronti (se il task vi si trova ancora inserito);
- Vengono inoltre gestiti i timer di fine capacità (se creati senza utilizzare il flag `CONTROL_CAP`) e quelli sulla deadline se presenti.

```
void task_end(LEVEL l, PID p);
```

Questa funzione implementa la terminazione di un task. In seguito alla chiamata di questa funzione il task è ufficialmente morto per il modulo e per l'intero sistema, pertanto ogni riferimento al task `p` (passato come parametro aggiuntivo) deve essere rimosso dalle strutture dati interne del sistema.

La funzione `task_end` è chiamata dopo le funzioni di cleanup ed i disruttori dei dati specifici dei task, ma prima dei distruttori dei modelli di risorsa gestiti all'interno del sistema.

In generale la funzione è richiamata in seguito alla terminazione naturale del task, oppure quando il task, una volta colpito a morte con una chiamata alla funzione `task_kill` o `pthread_cancel`, termina la sua vita in un punto di cancellazione.

Al momento della chiamata la funzione può assumere che il task si trovi nello stato `EXE`. Il tipico effetto della funzione è quello di togliere il task da eventuali code interne, eliminare gli eventi generati per tale task (deadline, capacità, ecc.), nonchè liberare il descrittore di task inserendolo nella coda `freedesc`. L'inserimento in tale coda indica al sistema che il descrittore può essere riutilizzato per la creazione di ulteriori task.

Se il Modulo implementa una gestione dei task con garanzia, l'inserimento del task all'interno della coda `freedesc` può essere posticipato al momento del decremento della banda occupata dal task, momento che in generale può avvenire dopo la morte effettiva del task (ad esempio, nei Moduli EDF e simili la banda viene decrementata a fine periodo).

```
void task_delay(LEVEL l, PID p, TIME tickdelay);
```

Questa funzione implementa un ritardo nell'esecuzione del task attivo. Occorre notare che la primitiva `task_delay`, che richiama questa task_call, ha un comportamento leggermente diverso dalla primitiva `nanosleep` presente nello standard POSIX (vedi Sezione 10.10).

La funzione accetta due parametri aggiuntivi costituiti dal task attualmente in esecuzione e dal tempo minimo, espresso in microsecondi, di sospensione del task.

Di seguito sono elencate le azioni tipiche che vengono compiute da questa funzione:

- Lo stato del task viene modificato per portarlo, nel caso in cui la schedulazione debba essere disattivata, in stati “di parcheggio” (quale, ad esempio, uno stato DELAY);

- Generalmente vengono gestite le capacità residue (ad esempio per implementare procedure di resource reclaiming);
- Il task viene rimosso da eventiuali code pronti (se il task vi si trova ancora inserito);
- Vengono inoltre gestiti i timer di fine capacità (se creati senza utilizzare il flag `CONTROL_CAP`) ma non quelli su eventuali deadline.
- Generalmente viene generato un evento che riattiva il task (ad esempio mettendolo in coda pronti, quando presente) alla fine del tempo trascorso.

### 8.3.3 Guest Calls

Di seguito sono elencate le Guest Calls presenti nel sistema. Occorre far notare alcuni aspetti importanti:

- Le Guest Calls non sono strettamente necessarie per il funzionamento del sistema. In effetti, se un Modulo non intende accettare task ospiti da altri Moduli, può semplicemente evitare di implementare tali funzioni, limitandosi a generare una eccezione di tipo `XUNVALID_TASK`, in modo tale da poter accorgersi dell'accaduto (in ogni caso non è consigliabile impostare il puntatore a funzione nel descrittore ad un valore `NULL`).
- Accettare come Guest un Modello di Task implica che il Modulo deve limitarsi a gestire solamente le caratteristiche che sono "implicitamente correlate" con il tale modello. Ad esempio, se un modello ha una deadline, il Modulo gestira' l'evento di deadline generando l'eccezione corretta, e così via. Di solito i Modelli usati per i Guest non posseggono

indicazioni relative al tempo di calcolo, che pertanto deve essere gestito dal Modulo che possiede realmente il task.

- La chiamata delle Guest Calls non è direttamente collegata allo scatenarsi di un evento per un task. Il caso tipico è quello in cui un task venga inserito in un altro Modulo tramite un Modello di Task che implementa una singola slice di esecuzione. Ad esempio, allo scattare di un evento di *endcycle* sarà richiamata la funzione `task_endcycle` che potrà richiamare la funzione `guest_end` per comunicare al livello Master che il task inserito viene "sganciato" dal Modulo Master.

- Quando un Modulo inserisce un task in un Modulo Master ne perde il controllo diretto; lo stato del task può essere così modificato dal Modulo Master in modo da gestire gli eventi correttamente (la notifica degli eventi avviene naturalmente con le Guest Calls). Un Modulo riacquisisce la possibilità di gestire un task solo quando il task viene "estratto" dal Modulo Master mediante una `task_end`.

- Non è possibile garantire a priori che un Modulo chiami le Guest calls nel modo corretto. Un Modulo di Schedulazione può comunque ipotizzare che le Guest Calls siano richiamate nel modo corretto.

Di seguito sono elencate tutte le Guest Calls. Il significato delle funzioni ed il loro comportamento è simile alle corrispondenti Task Calls, pertanto saranno indicate le sole differenze con queste ultime.

```
int guest_create(LEVEL l, PID p, TASK_MODEL *m);
```

La funzione viene utilizzata per inserire un task all'interno del Modulo. Il task inserito deve essere già stato creato mediante una primitiva `task_create`. La funzione tipicamente predispone il task ad essere attivato tramite una `guest_activate`. Tutte le informazioni utili passate con il Modello di Task devono essere registrate all'interno del Modulo, ovvero non esistono nel descritore di Task campi come `pclass` dove poter registrare le informazioni di un task inserito come Guest. Notare inoltre come il task dato in gestione al modulo rimanga un task di proprietà del Modulo che sta inserendo il task stesso[11].

`void guest_detach(LEVEL l, PID p);`

Questa funzione è richiamata da un Modulo quando per qualche motivo non è stato possibile inizializzare correttamente il guest task. La semantica della funzione è identica alla `task_detach`.

`void guest_dispatch(LEVEL l, PID p, int nostop);`

Questa funzione viene solitamente chiamata all'interno della `task_dispatch` per indicare al Modulo che un task inserito come guest sta subendo un evento di dispatch. La semantica della funzione è identica alla `task_dispatch`.

`void guest_epilogue(LEVEL l, PID p);`

Questa funzione è richiamata da un Modulo per indicare che un task inserito all'interno del Modulo sta terminando una slice di esecuzione. Questa funzione può venire richiamata da varie task calls, tutte le volte che il comportamento che si vuole ottenere dal task è simile a quello della `task_epilogue`.

`void guest_activate(LEVEL l, PID p);`

Questa funzione viene tipicamente richiamata subito dopo la `guest_create`, per attivare un task all'interno del Modulo Master. La funzione non deve

[11] In realtà nulla vieta che il Modulo che chiama le guest calls abbia a sua volta ricevuto il task che inserisce da un terzo Modulo. Con approcci di questo tipo possono essere implementati algoritmi di scheduling quali, ad esempio, il Bandwidth Sharing Server (BSS) [LBA98].

leggere il campo `request_time` del descrittore di task, in quanto la sua esecuzione non è collegata direttamente all'evento activate.

```
void guest_extract(LEVEL l, PID p);
```

La funzione implementa il bloccaggio del task `p` inserito come guest task su di una generica primitiva di sincronizzazione non implementata con il meccanismo degli shadow. La semantica della funzione è identica alla `task_extract`.

```
void guest_insert(LEVEL l, PID p);
```

L'effetto della chiamata di questa funzione è duale rispetto alla funzione `guest_extract`. La semantica della funzione è identica alla `task_insert`.

```
void guest_endcycle(LEVEL l, PID p);
```

Questa funzione implementa la fine di una attivazione del task `p` passato come parametro aggiuntivo. La semantica della funzione è identica alla `task_endcycle`.

```
void guest_sleep(LEVEL l, PID p);
```

Questa funzione implementa la fine di una attivazione del task `p` passato come parametro aggiuntivo. La semantica della funzione è identica alla `task_sleep`.

```
void guest_end(LEVEL l, PID p);
```

Questa funzione implementa la terminazione del guest task all'interno del Modulo Master. Non è detto comunque che il task sia realmente terminato. Pertanto la funzione non può in generale assumere che il task si trovi nello stato `EXE`. Il tipico effetto della funzione è quello di togliere il task da eventuali code interne ed eliminare gli eventi generati per tale task (deadline, capacità, ecc.), ma non quello di liberare il descrittore di task inserendolo nella coda `freedesc`.

```
void guest_delay(LEVEL l, PID p, TIME tickdelay);
```

Questa funzione implementa un ritardo nell'esecuzione del task passato come parametro. Si può supporre che il task in questione sia il task in esecuzione. La semantica della funzione è identica alla `task_delay`.

## 8.4 Funzioni di registrazione

Un Modulo, oltre a fornire le funzioni che implementano la propria interfaccia verso il Kernel Generico, deve fornire anche adeguate funzioni che hanno il compito di inizializzare in modo appropriato le strutture dati del Modulo e del Kernel Generico stesso.

Di solito la inizializzazione di un modulo è formata da 4 parti distinte:

- La prima parte riguarda l'allocazione di un descrittore di livello da utilizzare per inizializzare l'interfaccia del Modulo;
- La seconda parte riguarda l'inizializzazione della interfaccia generica di un Modulo di Schedulazione;
- La terza parte riguarda l'inizializzazione delle strutture dati private del Modulo, che comprende eventualmente la registrazione di una funzione di inizializzazione;
- La quarta parte riguarda tipicamente l'esecuzione della funzione di inizializzazione eventualmente registrata nella parte precedente.

Le prime tre parti si trovano all'interno di una apposita funzione fornita dai Moduli e chiamata dalla funzione `__kernel_init__`. Tale funzione viene richiamata prima che il kernel vada in modalità multitasking, pertanto quando risulta necessario creare task o invocare primitive di sistema per inizializzare correttamente un Modulo (si pensi al modulo che deve creare il task dummy o deve creare il task che ha il compito di chiamare la funzione main()) è necessario includere la quarta parte che sarà opportunamente chiamata non appena il sistema sarà andato in modalità multitasking (vedi Sezione 10.2).

## 8.5 Convenzioni nella scrittura di un modulo

In questa Sezione sono indicate alcune convenzioni seguite nello sviluppo dei Moduli di Schedulazione, utili per comprendere l'approccio utilizzato nello sviluppo di tali Moduli nonchè utili per scrivere Moduli in linea con gli altri.

- I codici di versione dei Moduli distribuiti con il Kernel si trovano nel file `include/modules/codes.h`;
- Un Modulo è costituito fondamentalmente da un solo file `.h` ed un solo file `.c`. I file `.h` dei moduli distribuiti con il Kernel si trovano nella directory `include/modules`, mentre i file `.c` si trovano nella directory `kernel/modules`. Se un Modulo è costituito da più file si consiglia di commentare adeguatamente il fatto all'inizio dei files del Modulo;
- Le funzioni di registrazione fornite con i Moduli registrano **UN SOLO** descrittore di livello o di risorsa. In questo modo si può conoscere il numero di livello a cui si registra un Modulo semplicemente ispezionando il file di inizializzazione;
- I Modelli di Task utilizzati dai Moduli distribuiti con il Kernel si trovano all'interno del file `include/kernel/model.h`;
- I nomi delle funzioni interne dei Moduli devono essere definite come `static` e devono essere nella forma `NOMEMODULO_NOMEFUNZIONE`, dove `NOMEMODULO` è il nome senza estensione del file `.c` in cui si trova il codice delle funzioni del Modulo;

- I prototipi delle funzioni esportate da un Modulo (funzioni di registrazioni più funzioni accessorie) devono essere incluse all'interno del file `.h`. Le funzioni accessorie di un Modulo tipicamente riceveranno come primo parametro il numero del livello del Modulo di appartenenza;

- I Moduli possono essere creati in più istanze all'interno di un file di configurazione, quindi i Moduli non devono fare affidamento su variabili globali.

La scrittura di un Modulo di Schedulazione richiede in generale la creazione di nuovi files. Per rendere più facile la distribuzione e l'integrazione di tali Moduli all'interno del Kernel si consiglia di non effettuare alcuna modifica ai file inclusi nella distribuzione standard ma di procedere come segue:

- Un nuovo Modulo di Schedulazione deve essere costituito da un solo file `.h` ed un solo file `.c`;

- Assieme al Modulo di Schedulazione devono essere forniti:

  - un file di inizializzazione (simile a quelli presenti nella directory `kernel/init`) che illustri come tale Modulo deve essere inizializzato;
  - almeno un programma di test che illustri le funzionalità del Modulo;

- I nuovi Modelli di Task, le nuove `pclass`, i codici di versione utilizzati dai nuovi Moduli ed eventuali eccezioni devono essere mantenuti all'interno del file `.h` del Modulo. Le costanti utilizzate devono possedere identificatori non utilizzati all'interno della distribuzione utilizzata[12].

[12] Occorre notare come tali codici possono necessitare di cambiamenti a seconda dell'ambiente in cui i nuovi Moduli si andranno a trovare (si pensi alla possibilità di mescolare assieme Moduli di Schedulazione non scritti per essere gestiti assieme).

Quando il Modulo di Schedulazione verrà integrato all'interno della distribuzione ufficiale i codici indicati nel file `.h` del Modulo saranno inseriti all'interno dei file correnti della distribuzione.

# Chapter 9

# I Moduli di Gestione delle Risorse

In questo capitolo viene descritta nel dettaglio l'interfaccia di un generico Modulo di Gestione delle Risorse. In particolare viene descritta la semantica delle funzioni che ne compongono l'interfaccia e vengono descritti gli approcci utilizzati per gestire i Protocolli di Accesso a Risorse Condivise. Per le informazioni architetturali fare riferimento alle Sezioni 4.3 e 4.4.

## 9.1 Interfaccia dei Moduli di Risorsa

Di seguito sono descritte le funzioni che caratterizzano l'interfaccia di un Modulo di Gestione delle Risorse. L'approccio utilizzato per la definizione dell'interfaccia è molto simile a quello seguito per i Moduli di Schedulazione (vedi Sezioni 8, 8.3 e 8.5). Per ogni funzione saranno fornite indicazioni generali riguardanti il suo utilizzo, oltre ad esempi riguardanti possibili implementazioni.

L'interfaccia di un Modulo di Gestione delle Risorse è comunque meno complessa di quella dei Moduli di Schedulazione: in pratica vengono gestiti solamente gli eventi *create* ed *end* del ciclo di vita di un task. Questa scelta è dovuta al fatto che molti algoritmi di gestione delle risorse non entrano direttamente nel merito della schedulazione del sistema[1].

Tutte le funzioni dei Moduli di Gestione delle Risorse sono richiamate ad **interruzioni disabilitate**.

```
void resource_status();
```

Questa funzione non ha parametri aggiuntivi e tipicamente realizza la stampa a video di informazioni statistiche riguardanti il Modulo. Il comportamento della funzione è analogo a quello della funzione `level_status` dei Moduli di Schedulazione.

Questa funzione è stata direttamente ereditata dalle precedenti versioni del Kernel Hartik, e probabilmente subirà alcune modifiche nelle versioni future del Kernel.

```
int level_accept_resource_model(RLEVEL l, RES_MODEL *r);
```

Questa funzione riceve come parametro aggiuntivo un puntatore ad una struttura `RES_MODEL`. La funzione è richiamata dalla primitiva `task_create` in quanto al momento della creazione occorre ricercare un Modulo di Gestione delle Risorse capace di gestire un eventuale Modello di Risorsa passato con la primitiva.

La funzione deve ritornare un valore `0` se il Modulo di Gestione delle Risorse può gestire un determinato modello, `-1` altrimenti.

Si ricorda che un Modulo di Gestione delle Risorse può accettare anche più di un Modello di Risorsa. Il reale Modello passato nel puntatore può essere dedotto dal valore del campo `rclass` della struttura `RES_MODEL`. Se un Modulo di Gestione delle Risorse accetta una certa `rclass`, allora lo stesso Modulo deve accettare il valore `rclass|l` (cfr. Sezione 6.3).

[1]Si può anche far notare come un algoritmo di gestione delle risorse che modifica profondamente la schedulazione del sistema è in pratica assimilabile più ad un Modulo di Schedulazione che ad un Modulo di Gestione delle Risorse. Eventuali approcci ibridi possono essere implementati richiedendo che l'implementazione sia costituita da due Moduli intercomunicanti.

```
void (*res_register)(RLEVEL l, PID p, RES_MODEL *r);
```

Questa funzione viene richiamata dal Kernel Generico all'interno della primitiva `task_create` per registrare un Modello di Risorsa all'interno di un Modulo di Gestione delle Risorse. La funzione accetta come parametri l'indice del descrittore allocato dal Kernel per il nuovo task ed uno dei Modelli di Risorsa passati tramite la primitiva di creazione.

Il Kernel Generico garantisce che il Modello di Risorsa passato in questa funzione appartiene ai Modelli di Risorsa gestibili dal Modulo[2]. Il valore ritornato dalla funzione è `0` se il Modulo può gestire la richiesta, `-1` se il task non può essere creato in quanto il Modulo di Gestione delle Risorse non può garantire le richieste[3]. La funzione ha il compito di inizializzare le strutture dati locali al Modulo con i parametri di Qualità del Servizio passati all'interno del Modello di Risorsa.

```
void (*res_detach)(RLEVEL l, PID p);
```

La chiamata di questa funzione da parte del Kernel Generico segnala al Modulo di Gestione delle Risorse che il task indicato nel parametro aggiuntivo p è ufficialmente terminato, e pertanto tutte le strutture dati del Modulo devono tenerne conto, eliminandone ogni riferimento.

Questa funzione viene chiamata (*indipendentemente dal fatto che un task abbia registrato o meno un Modello di Risorsa*) nei due casi seguenti:

- La primitiva `task_create` fallisce per qualche motivo indipendente dai voleri del Modulo; in questo caso la funzione viene richiamata prima della funzione `task_detach` del Modulo di Schedulazione.

- Il task termina in modo corretto; in questo caso la funzione viene richiamata prima della funzione `task_end` del Modulo di Schedulazione.

[2] In effetti, il Kernel Generico chiama la funzione dopo aver verificato tramite la `level_accept_resource_model` che il Modulo possa accettare il Modello di Risorsa.

[3] Ritornare -1 implica anche che la procedura di garanzia per i Moduli di Schedulazione non viene richiamata; al suo posto viene invece richiamata una `task_detach`.

In pratica questa funzione corrisponde ai comportamenti delle funzioni dei Moduli di Schedulazione `task_detach` e `task_end`. Questo fatto è corretto in quanto un Modulo di Gestione delle Risorse non entra mai nel merito della schedulazione del task, ma solo nel merito della sula creazione e terminazione. In questa ottica quindi non interessa se la terminazione di un task è avvenuta in modo normale oppure se il task non è stato proprio creato per altri motivi.

## 9.2 Implementazione dei protocolli di accesso a risorse condivise

L'interfaccia esportata dai Moduli di Gestione delle Risorse viene utilizzata anche dai Moduli che implementano protocolli di accesso a risorse condivise.

Il problema affrontato nello sviluppo dei Moduli è stato quello di progettare delle primitive di sistema che fossero indipendenti dal protocollo utilizzato e, soprattutto, indipendenti a run-time dai protocolli effettivamente registrati nel sistema.

### 9.2.1 Approccio utilizzato

L'approccio utilizzato per risolvere i problemi citati al paragrafo precedente è stato quello di estendere la interfaccia dei Moduli di Gestione delle Risorse, in modo tale da poter implementare sia i protocolli classici sia quelli che richiedano il passaggio di parametri aggiuntivi alla creazione dei mutex o dei task[4] .

L'estensione della interfaccia è stata resa possibile dall'approccio modulare utilizzato nello sviluppo dei Moduli di Gestione delle Risorse. Utilizzando tecniche di progettazione object oriented si può descivere la situazione con il diagramma in Figura 9.1.

[4]Tali parametri aggiuntivi sono, ad esempio, il ceiling di un mutex per i mutex Priority Ceiling, il livello di preemption per i task che utilizzano SRP, i mutex utilizzati da un determinato task, ecc.

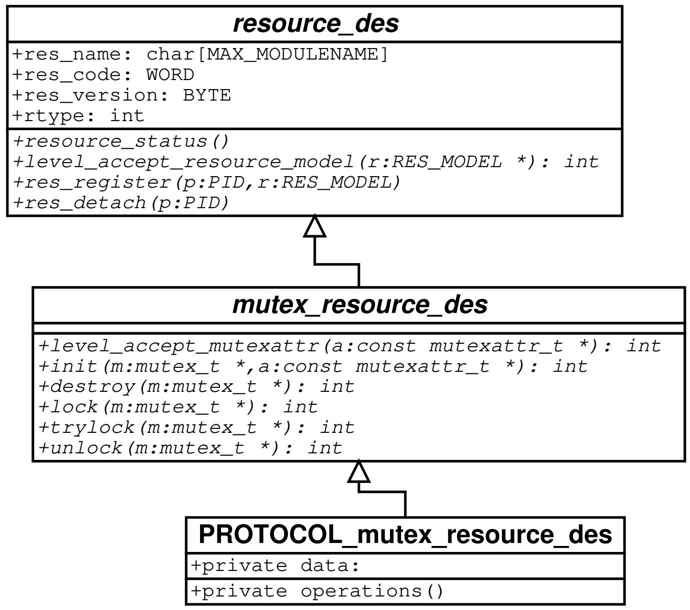


Figure 9.1: Diagramma UML che esemplifica la gerarchia di classi dei Moduli che implementano protocolli di accesso a risorse condivise.

Se paragoniamo l'interfaccia di un Modulo di Gestione delle Risorse ad una classe base di un generico linguaggio Object Oriented si può dire che i Moduli che implementano i protocolli di accesso ereditano la propria interfaccia da una classe base che a sua volta è derivata da quella dei Moduli di Gestione delle Risorse.

In pratica i Moduli vengono visti dal Kernel Generico come dei veri e propri Moduli di Gestione delle Risorse. Quando un mutex viene inizializzato viene effettuata una ricerca[5] per trovare il Modulo di Gestione delle Risorse che effettivamente estende l'interfaccia nel modo corretto[6] ed implementa il protocollo richiesto.

[5]Simile a quella fatta nella primitiva `task_create` per i Modelli di Task.

[6]Per sapere che un Modulo ha esteso l'interfaccia dei Moduli di Gestione delle Risorse viene messo a disposizione il campo `rtype` (vedi Sezione 7.2.3).

```
typedef struct {
   RLEVEL mutexlevel;
   int use;
   void *opt;
} mutex_t;
```

Figure 9.2: Struttura `mutex_t`.

### 9.2.2 I mutex

Per quanto riguarda i mutex essi vengono memorizzati nella struttura `mutex_t` illustrata in Figura 9.2. La dichiarazione può essere trovata anche nel file `include/kernel/descr.h`. Essa contiene i seguenti campi:

**mutexlevel** indica il livello in cui è registrato il Modulo di Gestione delle Risorse che estende l'interfaccia per implementare il protocollo utilizzato dal mutex.

**use** indica se il mutex è attualmente utilizzato o meno all'interno delle primitve di sincronizzazione delle variabili condition.

**opt** questo campo è un puntatore ad una struttura che il Modulo può allocare dinamicamente per contenere parametri dipendenti dal protocollo[7].

### 9.2.3 Estensione all'interfaccia

A questo punto è possibile descrivere le funzioni che costituiscono l'estensione dell'interfaccia dei Moduli di Gestione delle Risorse. Le funzioni proposte in pratica coprono tutte le azioni che riguardano direttamente i mutex all'interno del sistema.

[7] Tali parametri non possono essere memorizzati internamente al Modulo in quanto i mutex non sono allocati staticamente come i task o i semafori.

Le funzioni `level_accept_mutexattr` e `init` sono le uniche ad essere richiamate ad interruzioni disabilitate. Le altre funzioni devono provvedere autonomamente a tale disabilitazione, in quanto non esiste una regola generica per tali tipi di funzioni (a seconda del protocollo utilizzato alcune funzioni possono o meno essere bloccanti).

```
int level_accept_mutexattr(RLEVEL l, mutexattr_t *a);
```

Questa funzione viene chiamata all'interno della primitiva di inizializzazione dei mutex per verificare se un modulo possa o meno gestire mutex di un determinato protocollo. La funzione riceve un parametro aggiuntivo che rappresenta l'attributo di mutex passato con la inizializzazione.

La funzione ritorna un valore `0` se il Modulo può gestire il protocollo a cui appartiene l'attributo passato come parametro; ritorna `-1` in caso contrario.

```
int init(RLEVEL l, mutex_t *m, mutexattr_t *a);
```

Questa funzione viene chiamata per inizializzare un mutex con un determinato protocollo. La funzione accetta due parametri aggiuntivi rappresentati dal mutex da inizializzare e dall'attributo di mutex contenente i parametri da utilizzare nella inizializzazione.

La funzione ritorna un valore `0` se l'inizializzazione del mutex è avvenuta con successo, un codice di errore altrimenti. I codici di errore ritornabili dalla funzione devono essere compatibili con quelli previsti per la funzione `pthread_mutex_init` dello standard POSIX.

```
int destroy(RLEVEL l, mutex_t *m);

int lock(RLEVEL l, mutex_t *m);

int trylock(RLEVEL l, mutex_t *m);

int unlock(RLEVEL l, mutex_t *m);
```

Queste funzioni si occupano di implementare le funzionalità previste per i mutex ed hanno una semantica identica alle corrispondenti funzioni dello standard POSIX. In particolare esse ricevono come parametro aggiuntivo un puntatore al mutex sul quale è richiesta la particolare funzionalità, e ritornano `0` in caso di successo, oppure un adeguato codice di errore in caso contrario.

Si ricorda nuovamente che queste funzioni devono provvedere autonomamente a gestire il cambio di contesto all'interno del sistema, in quanto non è possibile fornire una regola fissa per tali funzioni (ad esempio, la operazione di lock, che tipicamente può comportare la sospensione del task, non è mai bloccante se viene utilizzato un protocollo SRP).

# Chapter 10

# Il Kernel

In questo capitolo verranno affrontate una serie di problematiche che riguardano la implementazione del Kernel Generico. Gli obiettivi di questo capitolo sono quelli di fornire delle informazioni utili per poter poi comprendere ed analizzare il codice che costituisce il Kernel stesso. Il codice descritto in questo capitolo si trova nella directory `kernel`.

## 10.1 Task di sistema e task utente

Il Kernel Generico classifica i task presenti nel sistema utilizzando due flag del campo control del descrittore di Task. Si è resa necessaria una distinzione tra le varie tipologie di task in quanto l'ambiente operativo messo a disposizione dal Kernel è un ambiente monoprocesso multithread[1], e pertanto questo comporta che alcune attività del kernel svolte all'interno di task debbano essere protette da possibili comportamenti erronei dell'utente.

I flag interessati nella classificazione sono il flag `SYSTEM_TASK` ed il flag `NO_KILL`:

- Il flag `SYSTEM_TASK` è utilizzato per distinguere all'interno del sistema

[1] Ovvero, all'interno del sistema possono esistere un insieme di task che condividono la stessa memoria comune.

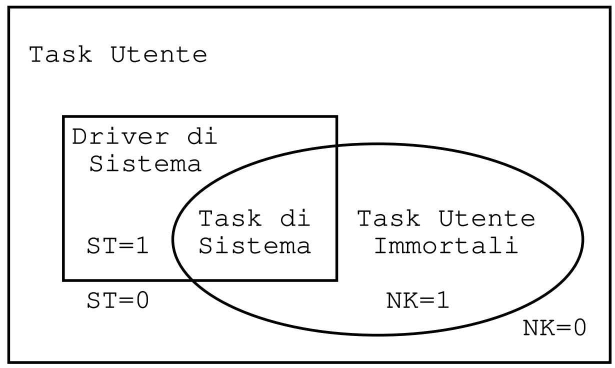


Figure 10.1: I quattro insiemi in cui sono suddivisi i task del sistema, con l'indicazione del valore dei flag `SYSTEM_TASK` (ST) e `NO_KILL` (NK).

| | `SYSTEM_TASK=0` | `SYSTEM_TASK=1` |
|---|---|---|
| `NO_KILL=0` | Task Utente | Driver di Sistema |
| `NO_KILL=1` | Task Utente Immortali | Task di Sistema |

Table 10.1: Valori dei flag `SYSTEM_TASK` e `NO_KILL` e suddivisione in quattro insiemi dei task del sistema.

i task utente dai task utilizzati internamente dal Kernel per i propri scopi;

- Il flag `NO_KILL` è utilizzato per disabilitare la possibilità di uccidere un task tramite una primitiva `task_kill` o `pthread_cancel`[2].

I valori che questi due flag possono assumere suddividono i task del sistema in quattro insiemi distinti (vedi Figura 10.1 e Tabella 10.1):

**Task Utente** Sono i task applicativi normalmente creati dall'utente;

**Task Utente Immortali** Sono task applicativi che l'utente vuole proteggere a priori da eventuali cancellazioni. In genere la vita di tali task non è determinante ai fini della terminazione del sistema (cioè il sistema

[2] In questo caso un task potrà uscire dal sistema solo alla sua terminazione corretta.

può effettuare uno shutdown indipendentemente dal fatto che tali task siano o meno terminati);

**Driver di Sistema** Sono task gestiti dal Kernel o da Librerie che implementano alcune caratteristiche indispensabili (ad esempio, il file system controlla le periferiche tramite dei task di questo tipo;

**Task di Sistema** Sono task non critici che però devono essere sempre presenti nel sistema, e dalla cui vita dipende il corretto funzionamento del sistema stesso. Ad esempio, un task di questo tipo è il task dummy.

La Terminazione del sistema può avvenire in modo esplicito da parte dell'utente tramite le chiamate alle funzioni `sys_end()` e `sys_abort()` oppure può venire generata automaticamente dal Kernel Generico.

Il Kernel Generico genera automaticamente la terminazione del sistema in due occasioni: quando tutti i Task Utente terminano oppure quando tutti i Driver di Sistema terminano.

Per quanto riguarda la seconda condizione si può far notare che per effettuare lo shutdown del sistema nel modo corretto è comunque necessario fare si che anche le librerie implementate con l'ausilio di Driver di Sistema terminino in modo corretto. La fase di terminazione del sistema verrà descritta con maggiore dettaglio nella Sezione 10.2.

## 10.2 Inizializzazione e Terminazione

In questa sezione presenteremo con maggiore dettaglio la struttura della funzione `__kernel_init__`, funzione richiamata dalla OS Lib subito dopo la partenza del sistema. Alcune informazioni al riguardo sono state fornite nella Sezione 5.9 limitatamente a quello che riguarda la interfaccia tra Kernel Generico ed OS Lib.

### 10.2.1 Disabilitazione delle interruzioni

La prima cosa che viene fatta all'ingresso di una funzione è una disabilitazione delle interruzioni. In effetti, alla partenza del sistema viene allocato dalla OS Lib un contesto, che chiameremo `global_context`. Questo è il contesto in cui sono eseguite la funzione `__kernel_init__` e buona parte delle funzioni da lei chiamate.

I contesti dei task saranno poi allocati in seguito tramite chiamate alla funzione `ll_context_create` (contenuta nella primitiva del Kernel Generico `task_create`); è necessario in ogni caso che il contesto di partenza viaggi ad interruzioni disabilitate in quanto non si vuole che esso possa essere interrotto in qualche modo nelle fasi successive. Le interruzioni verranno riabilitate automaticamente con il primo cambio di contesto.

### 10.2.2 Inizializzazione del gestore della memoria del Kernel Generico

Una volta disabilitate le interruzioni è possibile inizializzare il gestore della memoria dinamica. Esso deve essere inizializzato per primo in quanto utilizzato praticamente in tutte le fasi successive, a partire dalla registrazione dei Moduli presenti nel sistema.

### 10.2.3 Inizializzazione delle strutture dati statiche

La fase successiva è rappresentata dalla inizializzazione delle strutture dati del sistema allocate staticamente. In particolare, vengono inizializzati:

- i descrittori di task ed i corrispondenti task-specific data;
- la coda dei descrittori liberi;
- i vettori che contengono i puntatori ai descrittori dei Moduli;
- le strutture dati utilizzate per implementare i segnali POSIX;

- i vettori contenenti le funzioni di inizializzazione registrate tramite la funzione `sys_atrunlevel`.

### 10.2.4 Registrazione dei Moduli presenti nel sistema

Arrivati a questo punto della inizializzazione, il sistema si trova ad interruzini disabilitate con le strutture dati inizializzate. Quello che manca è la conoscenza della configurazione reale dei Moduli all'interno del sistema. Per questo motivo viene richiamata la funzione

```
TIME __kernel_register_levels__(void *arg)
```

Tale funzione ha il compito di richiamare in sequenza le funzioni di registrazione dei Moduli di Schedulazione e dei Moduli di Gestione delle Risorse.

Essa riceve come parametro un puntatore ad un'area di memoria contenente una struttura di tipo `multiboot`, dalla quale è possibile estrarre varie informazioni, tra cui i gli argomenti passati sulla riga di comando[3]. Tale parametro può essere utilizzato per modificare la struttura dei Moduli in modo dinamico all'inizio della vita del sistema.

Il valore ritornato dalla funzione rappresenta il tick di sistema nel caso in cui la OS Lib venga inizializzata con il timer periodico (vedi Sezione 5.7).

L'approccio scelto è stato quello di permettere all'utente di personalizzare a piacimento la configurazione dei Moduli. Per fare questo il Kernel Generico non dispone di una funzione `__kernel_register_levels__` predefinita; tale funzione deve essere invece indicata dall'utente che compila l'applicazione.

Per semplificare lo sviluppo delle applicazioni la distribuzione del Kernel dispone di una apposita directory (`kernel/init`) dove sono inserite una serie di funzioni di inizializzazione di esempio.

All'interno della funzione di registrazione (del Kernel e dei Moduli) possono essere utilizzate le seguenti funzioni:

[3] A tal proposito è possibile utilizzare la funzione `__compute_args__` descritta nel file `include/kernel/func.h`.

- Le funzioni di gestione della memoria dinamica (descritte nella Sezione 7.6.3);
- La funzione `sys_atrunlevel` utilizzabile per registrare funzioni di inizializzazione e/o di terminazione;
- Le funzioni della Libreria C esportate dalla OS Lib;
- Le funzioni per stampare messaggi a video, quali ad esempio `printk` e `kern_printf`.

**Non possono** invece venire utilizzate tutte le altre primitive del Kernel Generico (**comprese la `task_create` e la `task_activate`**) e della OS Lib (vedi Sezione 10.2.6).

Per quanti fossero familiari con le versioni precedenti del Kernel, il corpo di questa funzione corrisponde più o meno alla prima parte della funzione `main()` (fino alla `sys_init` esclusa).

### 10.2.5 Inizializzazione della OS Lib

Una volta inizializzate tutte le strutture dati, il sistema inizializza la OS Lib richiamando le funzioni `ll_init` ed `event_init` con i parametri restituiti dalla funzione di registrazione dei Moduli.

### 10.2.6 Esecuzione delle funzioni di inizializzazione

A questo punto il sistema si ritrova con un solo contesto valido (il contesto `global_context`) e senza nessun task nè di utente nè di sistema (in quanto nessuno ha potuto ancora crearne qualcuno).

Per terminare la fase di inizializzazione è pertanto necessario eseguire (all'interno del `global_context`) eventuali funzioni di inizializzaione registrate dai Moduli presenti nel sistema. Essendo la OS Lib inizializzata, possono essere richiamate tutte le funzioni esportate da tale libreria. Possono inoltre essere richiamate le primitive `task_create` e `task_activate`.

Le funzioni di inizializzazione vengono utilizzate dai Moduli di Schedulazione per creare un insieme di task di partenza. I task che tipicamente vengono creati in questa fase sono due: il task `dummy` (a cura del Modulo di Schedulazione `dummy`[4]) ed il task che ha come corpo la funzione `__init__` (tale task è creato dal Modulo di Schedulazione Round Robin (`RR`)). Questa scelta non è assoluta, nel senso che l'unica cosa da garantire dopo le funzioni di inizializzazione è che non manchi mai un task da schedulare all'interno del sistema.

La funzione `__init__` è contenuta all'interno dei file di inizializzazione assieme alla funzione `__kernel_register_levels__`. Tale funzione rappresenta il corpo del primo task eseguito all'interno del sistema. Le azioni compiute da tale funzione sono generalmente due:

- la inizializzazione di alcuni device e librerie che per loro natura utilizzano primitive di sistema (quali, ad esempio, i semafori) e pertanto devono essere inizializzate all'interno di un task vero e proprio (non nel `global_context` dove sono richiamate le funzioni di inizializzazione). Esempi di tali librerie sono le porte di comunicazione, il driver della tastiera e del mouse;

- la chiamata della funzione `main()` (in modo tale che le applicazioni possano essere scritte secondo lo standard del linguaggio C)[5].

Per quanti fossero familiari con le versioni precedenti del Kernel, il corpo della funzione `__init__` corrisponde più o meno alla parte della funzione `main()` compresa tra la `sys_init` (esclusa) e la `sys_end` finale (esclusa).

[4] Questo modulo ha l'unico scopo di terminare la pila dei Moduli di Schedulazione in modo tale che esista sempre un task da schedulare.

[5] A tal proposito è possibile utilizzare la funzione `__call_main__` descritta nel file `include/kernel/func.h`.

### 10.2.7 Prima schedulazione e cambio di contesto

Una volta inizializzate le strutture dati necessarie e creati i task di partenza, è sufficiente rischedulare il sistema per far si che venga eseguito uno task, qualunque esso sia.

La rischedulzione implica anche il salvataggio del contesto globale. Non essendo il contesto globale un task gestito dal sistema, il sistema non tornerà mai più in modo autonomo ad eseguire in quel contesto.

Le uniche primitive che provocano un ritorno sul contesto globale sono la `sys_end` e la `sys_abort`, chiamate in occasione dello shutdown dell'intero sistema. Notare come una chiamata alla funzione `ll_abort` provochi l'uscita incondizionata dalla OS Lib, senza un ritorno al contesto globale e pertanto senza effettuare lo shutdown del sistema in modo corretto.

### 10.2.8 Esecuzione delle funzioni di shutdown

Una chiamata alla funzione `sys_end`, o `sys_abort`, o la terminazine dell'ultimo task utente provocano un cambio di contesto sul contesto globale.

A questo punto è necessario effettuare delle operazioni per fare in modo che il sistema termini in modo corretto (ad esempio, se il File System è inizializzato è necessario terminare le operazioni in corso e scrivere su disco i dati non ancora salvati).

Le operazioni variano naturalmente a seconda dei Moduli registrati. Per ovviare a questo fatto il Kernel predispone, tramite la primitiva di sistema `sys_atrunlevel`, la possibilità di indicare una serie di funzioni da richiamare per effettuare lo shutdown.

Le funzioni indicate tipicamente attivano dei task di recupero che si preoccupano di terminare correttamente il sistema. Tali funzioni devono essere brevi, in quanto non appena finite verrà nuovamente rischedulato il sistema per permettere ad eventuali librerie (come il FileSystem) implementate con il supporto di task esterni di terminare correttamente.

Se le funzioni di shutdown diventano eccessivamente lunghe si possono verificare effetti indesiderati che possono portare ad un blocco del sistema[6].

### 10.2.9 Richiesta di terminazione per i task utente

Una volta chiamate le funzioni di shutdown il sistema, per velocizzare il processo di chiusura, tenta di uccidere tutti i task utente. Essendo la cancellazione spesso differita, non è detto che ciò provochi una morte repentina di tutti i task utente una volta ritornati in modalità multitasking.

### 10.2.10 Seconda schedulazione e cambio di contesto

Per effettuare una chiusura corretta del sistema è necessario a questo punto richiamare lo schedulatore e cambiare contesto. Per la seconda volta il sistema esce dal contesto globale. Il comportamento normale del sistema è il seguente:

- I task utente colpiti a morte durante il passo precedente dovrebbero lentamente morire;
- I task di sistema, informati adeguatamente dalle funzioni di shutdown dovrebbero finire i compiti in sospeso e terminare.

Il sistema ritornerà con successo sul contesto globale solo quando tutti i task di sistema saranno terminati o sarà chiamata una `sys_abort`. La `sys_end` in questa fase non ha alcun effetto.

[6]Un tipico caso è quello in cui al ritorno nella multiprogrammazione un task che stava utilizzando un device sfonda una deadline ed il Modulo di Schedulazione lo rende non più schedulabile. In questo caso la operazione sul device non può terminare e con esso tutto il sistema.

### 10.2.11 Esecuzione delle funzioni da chiamare prima della terminazione della OS Lib

Quando (si spera) tutti i task di sistema saranno terminati l'esecuzione ritornerà sul contesto globale. A questo punto verranno richiamate le funzioni registrate con la primitiva `sys_atrunlevel` richiamata con il parametro `RUNLEVEL_BEFORE_EXIT`.

Il compito di queste funzioni è quello di terminare l'opera di pulizia all'interno del sistema (ad esempio una delle classiche cose da fare è quella di riportare lo schermo in modalità testo).

### 10.2.12 Terminazione della OS Lib

A questo punto il sistema è pronto per terminare i propri compiti. Per questo motivo viene richiamata la `ll_end` che libera le strutture dati allocate con la `ll_init`. Da ora in poi potranno essere richiamate di nuovo solo le funzioni specificate nella Sezione 10.2.4.

### 10.2.13 Esecuzione delle funzioni da chiamare dopo la terminazione della OS Lib

Infine, il Kernel richiama le ultime funzioni registrate con la primitiva di sistyema `sys_atrunlevel`, le quali tipicamente possono fare ben poco, al limite la stampa di qualche messaggio augurale o di debug.

## 10.3 Creazione di un task e sua garanzia

La primitiva del Kernel Generico che si occupa di creare e garantire un nuovo task si chiama `task_createn`. Il prototipo di tale funzione è il seguente (il codice della funzione è contenuto nel file `kernel/create.c`):

```
PID task_createn(char *name, TASK (*body)(), TASK_MODEL *m, ...);
```

I parametri passati a tale funzione sono i seguenti:

`name` Nome simbolico del Task, utilizzato a fini statistici;

`body` Puntatore a funzione che indica il corpo del task;

`m` Puntatore ad un Modello di Task del nuovo task da creare

... Serie di puntatori a Modelli di Risorsa terminati con `NULL`.

La primitiva restituisce il numero del descrittore di task associato al nuovo task, o `NIL` se il task non può essere creato all'interno del sistema. Nell'ultimo caso la variabile `errno` assume un valore che indica la tipologia dell'errore[7].

Esiste poi una ridefinizione della primitiva chiamata `task_create` che come unica differenza accetta un unico Modello di Risorsa al posto dei "...". Questa ridefinizione risulta utile in quanto spesso non viene passato un numero elevato di Modelli di Risorsa.

I passi compiuti per effettuare correttamente la creazione e la garanzia di un nuovo task sono descritti di seguito.

Per prima cosa occorre ritrovare un descrittore di task inutilizzato. Tipicamente i descrittori iniutilizzati si trovano accodati nella apposita coda `freedesc`. Nella scelta del descrittore andranno scartati quelli utilizzati per mantenere il valore di ritorno dei task che, pur essendo terminati, attendono una sincronizzazione tramite una `task_join` (vedi Sezione 10.9).

A questo punto il descrittore viene allocato rimuovendolo dalla coda dei descrittori liberi e inizializzato con valori di default.

Fatto questo è necessario ricercare all'interno dei Moduli di Schedulazione registrati se esiste un Modulo di Schedulazione capace di gestire il Modello passato come parametro. La ricerca viene effettuata chiamando, a partire

[7] I codici di errore si trovano nel file `include/bits/errno.h`.

dal Modulo di livello 0, la funzione di interfaccia `level_accept_task_model`. Una volta trovato il Modulo giusto il task viene creato all'interno del Modulo tramite una chiamata alla funzione di interfaccia `task_create`.

Il passo successivo nella creazione del task è l'inizializzazione dei Modelli di Risorsa passati. Tale inizializzazione viene compiuta richiamando la funzione `level_accept_resource_model` sui Moduli di Gestione delle Risorse registrati nel sistema. Una volta trovato il Modulo giusto la risorsa viene registrata tramite una chiamata alla funzione `res_register`.

A questo punto tutti i componenti del sistema sono stati informati della Qualità del Servizio e dei parametri aggiuntivi richiesti dal nuovo task, e pertanto ora, e solo ora, è possibile tentare una garanzia del sistema. La garanzia non può essere richiamata prima della registrazione dei Moduli di Risorsa perchè in generale possono essere sviluppati Moduli "ibridi", cioè Moduli che vengono visti dal sistema sia come Moduli di Schedulazione che come Moduli di Gestione delle Risorse (si pensi ad un Modulo che registra due descrittori ed implementa EDF con supporto per SRP). Occorre quindi necessario richiamare la garanzia del sistema solo dopo che tutti i Modelli passati al momento della creazione (quello di Task più gli eventuali di Risorsa) siano stati registrati.

Se infine il task può essere garantito si può procedere alla allocazione della memoria per lo stack del task (se necessaria), alla creazione del contesto (tramite la chiamata della funzione `ll_context_create`), alla registrazione dell'evento per il tracer e all'eventuale conteggio del task (necessario per implementare la terminazione automatica del sistema).

Se uno qualsiasi dei passi precedenti fallisce il sistema viene riportato alla situazione precedente alla chiamata della primitiva (eventualmente chiamando se necessario le funzioni `task_detach` e `res_detach` dei Moduli).

Per quanto riguarda la Garanzia on-line del sistema, il Kernel Generico supporta una procedura di garanzia distribuita tra i vari Moduli di Schedulazione basata sul fattore di utilizzazione. In pratica il sistema richiama in cascata la funzione `level_guarantee`, passando come parametro aggiuntivo la banda lasciata libera dai moduli precedentemente garantiti. Tale procedura è implementata all'interno della funzione `guarantee()` contenuta nel file `kernel/kern.c`.

In effetti questo approccio permette di implementare in modo sufficientemente semplice la garanzia on-line di molti algoritmi di schedulazione[8] . Ovviamente non possono essere implementati in cascata algoritmi più complessi, quali ad esempio la garanzia del Deferrable Server, dell'algoritmo TB* [BS97] ed altri. In questi casi si possono utilizzare due strategie:

- Si garantiscono i task del sistema off-line, disabilitando la procedura di garanzia a run-time.
- Si sviluppano i Moduli che necessitano della garanzia come un unico Modulo di Schedulazione, piazzato al primo livello del sistema. In questo modo esso riceve e può controllare tutta la banda libera del sistema, e con essa, avendo tutte le informazioni sui task da garantire, può implementare l'algoritmo di garanzia in modo classico. Ovviamente con questo approccio si perdono tutti i benefici ottenuti con la Modularità degli algoritmi di Scheduling.

## 10.4 Attivazione di un task

Il Kernel Generico, a differenza dello standard POSIX, separa la creazione di un task e la sua garanzia dalla attivazione vera e propria. Il motivo di questa scelta è duplice, in quanto in letteratura molti risultati sono stati ottenuti per task che vengono attivati contemporaneamente all'inizio del major cycle; oltre a questo occorre considerare che la procedura di creazione può essere pesante a causa di procedure di garanzia, mentre l'attivazione è una attività molto più snella.

[8] Algoritmi che utilizzano garanzie di questo tipo sono, ad esempio, EDF, RM, Polling Server, Sporadic Server, CBS, TBS.

Le primitive che il Kernel mette a disposizione per attivare un task sono due:

`task_activate` Attiva un task singolo[9];

`group_activate` Attiva un gruppo di task in modo atomico[10].

Il Kernel Generico mette inoltre a disposizione un meccanismo che permette di congelare le attivazioni dei task. Tale meccanismo è stato inserito per permettere una implementazione modulare dei protocolli di gestione delle risorse condivise tipo SRP e simili.

Tale meccanismo utilizza il flag del campo control `FREEZE_ACTIVATION`, che indica lo stato di congelamento delle attivazioni, ed un campo del descrittore di Task, `frozen_activations`, che contiene il numero di attivazioni congelate dal Kernel Generico.

Sono infine definite le due seguenti primitive:

`task_block_activation` blocca le attivazioni esplicite dei task e ne attiva il conteggio nell'apposito campo. La funzione ritorna normalmente `0`, o `-1` se l'indice del task considerato non e' corretto.

`task_unblock_activation` rende di nuovo possibile riattivare un task. Ritorna `-1` se il task non aveva attivo il flag `FREEZE_ACTIVATION`, oppure il numero di attivazioni congelate. Se ci sono attivazioni congelate la primitiva NON effettua tali attivazioni.

I prototipi delle primitive presentate in questa Sezione sono indicati in Figura 10.2 e sono contenuti nei file `kernel/activate.c` e `kernel/blkact.c`.

[9] Questa primitiva può essere chiamata anche all'interno di un evento o all'interno del global_context (ossia durante le funzioni gestite con la primitiva `sys_atrunlevel`).

[10] Cioè il sistema viene rischedulato una sola volta quando tutti i task appartenenti ad un gruppo vengono attivati.

```
int task_activate(PID p);
int group_activate(WORD g);
int task_block_activation(PID p);
int task_unblock_activation(PID p);
```

Figure 10.2: Prototipi delle funzioni di attivazione.

## 10.5 Lo Scheduler

Cosa succede quando il sistema deve effettuare una rischedulazione, per scegliere un nuovo task da eseguire? I passi da compiere sono fondamentalmente tre:

- Se al momento della schedulazione stava eseguendo un task, occorre segnalare la fine della slice[11] per il task;
- Successivamente occorre ricercare il nuovo task da eseguire (scheduling);
- Infine bisogna mettere in esecuzione il task scelto (dispatching).

Tali passi sono implementati all'interno delle primitive di sistema e all'interno della funzione `scheduler()` contenuta nel file `kernel/kern.c`. Di seguito saranno analizzati i tre punti nel dettaglio.

### 10.5.1 Fine slice per il task in esecuzione

Per specificare quali sono le azioni da compiere alla fine di una slice del task in esecuzione è necessario comprendere quali sono le possibili motivazioni che comportano una rischedulazione.

[11]Con slice del task in esecuzione si intende in questo contesto l'intervallo di tempo che passa dal momento in cui un task viene messo in esecuzione al momento in cui viene eseguita una schedulazione del sistema.

A seconda della motivazione dovranno essere eseguite azioni diverse per far terminare la slice corrente. Per questo motivo il Kernel Generico prevede all'interno dell'interfaccia dei Moduli di Schedulazione diverse funzioni di terminazione di tale slice. I Motivi per cui può avvenire una rischedulazione sono indicati nel successivo elenco (la funzione dell'interfaccia è indicata tra parentesi):

1. Un nuovo task è divenuto attivo all'interno del sistema e quindi occorre verificare se si deve attuare una preemption (`task_epilogue`). Questa situazione può verificarsi in moltissimi casi, ad esempio:

    - viene attivato un nuovo task con una primitiva `task_activate` o `group_activate`;
    - viene liberata una risorsa e quindi attivato un task bloccata su di essa;
    - un task periodico viene riattivato all'inizio del proprio periodo;
    - un Driver di Sistema viene attivato per l'arrivo di una interruzione dal dispositivo che controlla;

2. Il task in esecuzione termina la propria capacità (`task_epilogue`);

3. Il task in esecuzione si blocca su di una primitiva di sincronizzazione (`task_extract`);

4. Il task in esecuzione termina una attivazione sospendendosi con la primitiva `task_endcycle` (`task_endcycle`);

5. Il task in esecuzione termina una attivazione sospendendosi con la primitiva `task_sleep` (`task_sleep`);

6. Il task in esecuzione si sospende momentaneamente con una primitiva `task_delay` (`task_delay`);

7. Il task in esecuzione termina o viene ucciso con una primitiva `task_kill` o `pthread_cancel` (`task_end`);

In generale non dovrà essere richiamata la sola funzione di interfaccia; la reale sequenza ha la seguente forma:

1. Lettura del tempo corrente all'interno della variabile `schedule_time`[12];

2. Calcolo della lunghezza temporale della slice appena terminata tramite l'utilizzo delle variabili `schedule_time` e `cap_lasttime`;

3. Aggiornamento delle informazioni riguardanti l'accounting dei tempi di calcolo (vedi Sezione 10.6);

4. Cancellazione dell'eventuale evento di capacità rimasto pendente;

5. Chiamata della funzione dell'interfaccia del Modulo di Schedulazione del Task che ha appena terminato la slice.

Per semplificare la scrittura delle primitive è stato seguito il seguente approccio: essendo la rischedulazione per preemption la più comune, la sequenza appena descritta terminante con una chiamata alla funzione `task_epilogue` è stata inclusa come prologo della funzione `scheduler()`. Tale prologo non viene eseguito se al momento della chiamata della funzione scheduler le variabili `exec` ed `exec_shadow` hanno un valore `NIL` (`-1`).

### 10.5.2 Scheduling

Una volta terminata correttamente la slice precedente è necessario scegliere un nuovo task da eseguire. L'algoritmo di scheduling generico implementato parte dal Modulo di Schedulazione presente a livello 0 chiamando la

[12]Tale variabile è utilizzata come riferimento temporale per l'istante di schedulazione. Notare come il Kernel Generico non separi il tempo di CPU passato ad eseguire codice utente e codice di sistema, ma tutto il tempo sia assegnato ai vari task.

funzione di interfaccia `level_scheduler`, scendendo man mano ai livelli inferiori quando un Modulo non ha task da schedulare. Il sistema suppone che esista **sempre** un task da schedulare[13].

Una volta trovato un task, viene richiamata la funzione `task_eligible` per verificare se il task scelto dallo scheduler è corretto. Se eventualmente il task risulta non essere corretto l'algoritmo di scheduling riprende come se fosse appena arrivato alla funzione `level_scheduler` che ha fornito il task "sbagliato" (vedi Sezione 8.3.2), rieseguendola.

### 10.5.3 Dispatching

Per trovare il task che veramente deve essere eseguito occorre fare ancora un piccolo passo, cioè occorre scorrere la catena degli shadow (vedi Sezione 4.4). Una volta trovato tale task viene richiamata su di esso la funzione `task_dispatch` della interfaccia dei Moduli di Schedulazione.

Se infine il task ha il bit `CONTROL_CAP` del campo control del descrittore di task settato viene generato un evento di fine capacità.

## 10.6 Statistiche sui tempi di esecuzione

Una delle caratteristiche del Kernel Generico è quella di supportare, tramite delle apposite primitive, la gestione delle statistiche dei tempi di calcolo dei task all'interno del sistema.

Questa caratteristica risulta molto utile in quanto le prestazioni di molti algoritmi proposti in letteratura dipendono in modo pesante da una stima precisa dei tempi di calcolo dei task all'interno del sistema. Avere a disposizione un insieme di primitive per la gestione delle informazioni principali riguardanti i tempi di calcolo dei task all'interno del sistema significa poter

[13]Per questo motivo viene fornito di serie con la distribuzione un Modulo di Schedulazione chiamato `dummy` che ha come compito quello di creare un task da utilizzare per garantire sempre la presenza di almeno un task da schedulare.

```
int jet_getstat(PID p, TIME *sum, TIME *max,
      int *n, TIME *curr);
int jet_delstat(PID p);
int jet_gettable(PID p, TIME *table, int n);
void jet_update_slice(TIME t);
void jet_update_endcycle();
```

Figure 10.3: Primitive per la gestione dei tempi di esecuzione e funzioni correlate utilizzate internamente dal Kernel Generico.

stimare con precisione i tempi di esecuzione e quindi poter sfruttare al meglio le prestazioni degli algoritmi di schedulazione implementati.

Per abilitare il Kernel Generico a registrare le informazioni sui tempi di esecuzione di un task occorre indicare le proprie intenzioni al momento della creazione del task stesso, tramite le apposite macro messe a disposizoone (vedi Sezione 6.2.1). Tali macro modificano il flag `JET_ENABLE` all'interno del campo `control` del descrittore di task.

Le informazioni mantenute dal Kernel Generico permettono di conoscere il tempo di esecuzione medio e massimo delle attivazioni del task, nonchè il tempo consumato dalla attivazione corrente ed il tempo consumato dalle ultime `JET_TABLE_DIM` attivazioni.

I prototipi delle primitive messe a disposizione dal Kernel Generico, assieme alle funzioni utilizzate internamente, sono descritte in Figura 10.3. Di seguito è riportata una descrizione delle primitive implementate:

`jet_getstat` Questa primitiva restituisce informazioni aggregate riguardo le statistiche dei tempi di esecuzione del task indicato come primo parametro. In particolare, le informazioni restituite sono registrate nei seguenti parametri:

  `sum` indica il tempo totale di esecuzione dal momento in cui il task è stato creato o dall'ultima chiamata alla funzione `jet_delstat`;

`max` indica il massimo tempo utilizzato da una istanza del task dal momento in cui è stato creato o dall'ultima chiamata alla funzione `jet_delstat`;

`n` indica il numero di istanza del task su cui sono stati calcolati i parametri `max` e `sum`;

`curr` indica il tempo di esecuzione consumato dalla istanza attualmente in esecuzione.

Se un parametro viene passato con valore NULL la corrispondente informazione non è restituita dalla primitiva. La funzione ritorna `0` se le informazioni fornite sono corrette, o `-1` se il task passato come parametro non esiste o tale task non ha il bit `JET_ENABLE` settato.

`jet_delstat` La primitiva imposta a `0` le informazioni attualmente mantenute all'interno del Kernel Generico per il task `p` passato come parametro. La funzione ritorna `0` se il task indicato esiste ed ha il flag `JET_ENABLE` settato, `-1` altrimenti.

`jet_gettable` La primitiva restituisce gli ultimi n tempi di esecuzione delle attivazioni del task indicato. Se il valore n indicato come parametro è minore di `0` verranno restituiti solo gli ultimi valori registrati dall'ultima chiamata alla funzione `jet_gettable`. Se il valore è maggiore di `0` verranno restituiti gli ultimi `min(n, JET_TABLE_DIM)` valori registrati. Il valore ritornato dalla funzione è `-1` se il task passato come parametro non esiste o tale task non ha il bit `JET_ENABLE` settato, oppure il numero di valori registrati all'interno della tabella passata come parametro. La tabella passata come parametro deve contenere almeno `JET_TABLE_DIM` elementi.

Le funzioni utilizzate all'interno delle primitive del Kernel sono invece le seguenti:

`jet_update_slice` aggiorna il tempo consumato dalla esecuzione corrente del task puntato dalla variabile `exec_shadow` di t microsecondi;

`jet_update_endcycle` aggiorna i tempi di esecuzione dell'ultima istanza indicando che l'istanza corrente del task puntato da `exec_shadow` è terminata.

## 10.7 La Cancellazione

La cancellazione di un task è probabilmente uno dei temi più delicati e complessi all'interno di un Kernel. Questo perchè la cancellazione di un task deve essere fatta correttamente in modo da non lasciare inconsistente lo stato delle risorse all'interno del sistema.

Lo standard POSIX prevede dei meccanismi di abilitazione e disabilitazione della cancellazione, oltre all'introduzione dei concetti di cancellazione asincrona e differita. Questi concetti assieme a molti altri, sono mantenuti all'interno di questo Kernel. In questa sezione non verranno descritte nel particolare le funzioni riguardanti la cancellazione di un task ma verranno discusse alcune scelte implementative.

Per maggiori informazioni sull'interfaccia delle funzioni correlate alla cancellazione si faccia riferimento allo standard POSIX. InTabella 12.2 sono riportati i nomi di alcune primitive correlate alla cancellazione che hanno un analogo (a parte il nome della funzione) nello standard citato.

I problemi più grossi nell'implementare la cancellazione dei task all'interno del Kernel Generico sono i seguenti:

- Il Kernel non ha un vero e proprio contesto o stack o livello di protezione in cui poter funzionare, ma funziona fondamentalmente con l'aiuto della disabilitazione delle interruzioni all'interno dei contesti dei task presenti nel sistema;

- Le funzioni di cancellazione dei singoli task devono essere eseguite all'interno del contesto del task che sta morendo, e quindi non è possibile uccidere un task rimanendo all'interno del contesto di un altro task (come succedeva invece con la precedente versione del Kernel);

- Il Kernel Generico deve poter astrarre dai punti di cancellazione presenti all'interno del sistema, anche perchè non è detto che il Kernel Generico all'interno della primitiva di kill abbia la possibilità di agire sulle strutture dati introdotte da una libreria particolare.

La soluzione a questi problemi è proposta nelle Sezioni seguenti.

### 10.7.1 La funzione task_makefree

Il fatto che le funzioni di cancellazione di un task debbano essere richiamate all'interno del contesto del task stesso ed il fatto che il Kernel Generico non abbia un contesto proprio comportano come conseguenza il fatto che un task per morire deve andare ad eseguire volontariamente una funzione appositamente predisposta (che all'interno del Kernel è chiamata `task_makefree`).

Tale funzione deve poter venire richiamata all'interno dei punti di cancellazione (in particolare viene richiamata nella funzione `task_testcancel` nel file `kernel/cancel.c`), nonchè alla terminazione del task (vedere la `task_create_stub` nel file `kernel/create.c`), ed ogni volta che il task viene schedulato (per testare una eventuale cancellazione asincrona).

I compiti della funzione `task_makefree` sono i seguenti:

- Controllare se qualcuno sta attendendo la terminazione del task all'interno di una `task_join`;

- Verificare se il task che sta morendo sta attualmente utilizzando una risorsa la cui mutua esclusione è implementata con il meccanismo degli shadow (in questo caso viene sollevata una eccezione);

- Chiamare eventuali funzioni di cleanup;
- Invocare i distruttori dei dati specifici dei task (i thread specific data del POSIX, per intenderci);
- Liberare il contesto[14] e la memoria allocata per lo stack;
- Chiamare le funzioni `task_end` sul Modulo di Schedulazione che possiede il task nonchè le funzioni `res_detach` sui Moduli di Gestione delle Risorse registrati nel sistema;
- Verificare infine che il task che sta terminando non sia l'ultimo di quelli appartenenti ad una delle classi di task per cui è prevista la terminazione del sistema (vedi Sezione 10.1).

### 10.7.2 La registrazione dei punti di cancellazione

L'ultimo problema da risolvere è quello dell'indipendenza del Kernel Generico dai punti di cancellazione. In effetti quello che si vuole veramente è che le funzioni che implementano i punti di cancellazione possano essere scritte e modificate senza dover andare ad incidere sulle primitive che effettivamente devono cancellare un task. Per implementare questo disaccoppiamento si è proceduto nel seguente modo:

- Il bloccaggio di un task su un punto di cancellazione viene implementato tramite la funzione di interfaccia `task_extract`;
- Un stato di un task bloccato su di un punto di cancellazione viene modificato per assumere un valore visibile a livello di Kernel (di solito tali nomi iniziano con la parola WAIT);

[14] Notare il fatto che il contesto liberato è quellIo in esecuzione. Questo non è un problema in quanto la `task_makefree` è eseguita ad interruzioni disabilitate e nessuno può utilizzare le aree di memoria appena liberate.

- Le funzioni che implementano punti di cancellazione, alla loro prima esecuzione, si registrano all'interno del sistema tramite una primitiva appositamente predisposta, `register_cancellation_point` (definita nel file `kernel/kill.c`). La primitiva in questione accetta un puntatore ad una funzione il cui valore di ritorno è `1` se un particolare task indicato nel parametro è bloccato su di un determinato punto di cancellazione.

- La funzione che ha il compito di uccidere un task, oltre a settare un apposito flag (il flag `KILL_REQUEST` del campo control del descrittore di task), richiama le funzioni registrate dai vari punti di cancellazione per verificare se un task si trovi bloccato all'interno di uno di tali punti di cancellazione. Nel caso affermativo, la funzione registrata ha il compito di riattivare il task bloccato, tipicamente chiamando la funzione `task_insert`.

- La struttura dei punti di cancellazione deve essere tale da far si che un task risvegliato a causa di una kill esegua appena risvegliato (o comunque sia prima possibile) un test per verificare una eventuale kill (tale test di solito viene fatto inserendo nel codice direttamente la primitiva `task_testcancel`).

Questo approccio ha alcuni vantaggi, tra cui quello non indifferente di permettere di isolare tutto il codice di un punto di cancellazione all'interno di un solo file (invece di spezzettarlo anche nel file che implementa la cancellazione), rendendo in questo modo più modulare e manutenibile il sistema.

### 10.7.3 Cleanups e Thread Specific Data

Lo standard POSIX prevede la possibilità di indicare tramite le primitive `pthread_cleanup_push` e `pthread_cleanup_pop` delle funzioni da eseguire nel caso in cui il task subisca una kill all'interno di una determinata sezione di codice delimitata da una coppia push/pop di queste funzioni.

L'implementazione di queste due funzioni è stata realizzata tramite una macro simile a quella consigliata all'interno dei Rationales dello standard POSIX. Tale implementazione si trova all'interno dei files `include/pthread.h` e `include/kernel/func.h`.

Il Kernel Generico dispone inoltre di un supporto per i Thread Specific Data dello standard POSIX. L'implementazione di tali primitive non è molto complicata e può essere ritrovata nel file `kernel/keys.c`.

## 10.8 I Segnali

Lo standard POSIX prevede una gestione abbastanza complessa di entità chiamate segnali, ovvero di interruzioni software con particolari caratteristiche e di particolari tipologie. La distribuzione del Kernel comprende una implementazione dei segnali POSIX direttamente derivata dal Flux OSKit [FBB+97], comprendente, oltre alla parte dei segnali standard, anche le estensioni real-time e le estensioni per i thread.

Il problema maggiore riscontrato nel porting del codice è stato quello di gestire in modo corretto i due seguenti aspetti:

- l'implementazione delle funzioni interrompibili dai segnali;
- il delivery corretto dei segnali.

Il primo problema è stato risolto prevedendo una registrazione delle funzioni interrompibili dai segnali in modo analogo a quanto fatto per i punti di cancellazione. Ogni volta che un segnale viene generato, viene controllata la presenza di task bloccati su primitive interrompibili, svegliando tali task quando necessario. La funzione di registrazione in questo caso si chiama `register_interruptable_point` e si trova all'interno del file `kernel/signal.c`.

Il secondo problema è stato affrontato inserendo la funzione denominata `kern_deliver_pending_signals` (definita nel file `kernel/signal.c` e responsabile del delivery corretto dei segnali) all'interno della macro che si occupa del cambiamento di contesto, la macro `kern_context_load` (definita nel file `include/kernel/func.h`). La chiamata della funzione avviene dopo il ritorno da un cambio di contesto, in modo tale che ogni volta che un task viene rischedulato venga eseguito il delivery dei segnali.

Rispetto alla implementazione dell'OS Kit i segnali hanno subito le seguenti, piccole, modifiche:

- è stata implementata la prenotazione delle strutture dati utilizzate per contenere i segnali in modo tale da poter implementare senza problemi i timer realtime previsti dallo standard POSIX;
- è stato previsto il settaggio del mittente di un segnale (POSIX permette di specificare se un segnale è stato generato da una promitiva utente, da un timer, da una message queue o da una operazione di input/output asincrono).

## 10.9 Join su di un task

Lo standard POSIX specifica che i thread del sistema sono definiti tramite funzioni che ritornano un valore `void *` che può essere recuperato, se il task è di tipo *joinable*, tramite una chiamata alla primitiva `pthread_join` o `task_join`.

In questa sezione verrà descritta l'implementazione della primitiva `di sistema task_join` e delle modifiche che tale primitiva ha comportato in varie parti del Kernel Generico.

Innanzitutto l'informazione sul tipo di task (joinable o detached) è mantenuta all'interno del flag `TASK_JOINABLE` del campo `control` del descrittore di task.

In generale i thread POSIX nascono nello stato joinable e poi possono essere declassati a detached. Il Kernel Generico segue questa via per l'implementazione della `pthread_create`, mentre invece l'attributo di default per i Modelli di Task è detached.

La primitiva `task_join` è la corrispondente primitiva del Kernel Generico che implementa la primitiva POSIX `pthread_join`. Essa è un punto di cancellazione e come tale si comporta registrandosi all'interno del Kernel.

Il problema principale riscontrato nell'implementazione di questa primitiva è il fatto che il descrittore di un task terminato correttamente non può essere riutilizzato fino a quando non vi è stata effettaua una join. Il problema si pone in quanto costringe i Moduli di Schedulazioni a conoscere l'implementazione interna della primitiva, cosa che complica abbastanza i Moduli in special modo se tali Moduli effettuano garanzie particolari sui task.

L'implementazione è riuscita comunque ad aggirare il problema nel seguente modo:

- I Moduli di Schedulazione prescindono dalla effettiva tipologia del task (joinable o detached) e si comportano in modo tale da inserire nella coda freedesc il descrittore del task non appena esso si liberi (un descrittore in generale può liberarsi anche molto dopo la terminazione del task a causa delle procedure di garanzia dei Moduli di Schedulazione);

- All'inizio della funzione `task_makefree` viene controllato se il task è di tipo *joinable*; in questo caso viene settato un bit, il bit `WAIT_FOR_JOIN` all'interno del campo control del descrittore del task che sta terminando. Il contesto e lo stack del task terminato ma in attesa di join vengono rilasciati;

- Una eventuale `task_create` che tenti di allocare il descrittore di un task terminato in attesa di join, il cui descrittore è stato inserito all'interno della coda freedesc da parte del suo Modulo di Schedulazione, ha come risultato quello di "scartare" il descrittore settando un altro bit, il bit

`DESCRIPTOR_DISCARDED`, all'interno del campo `control` del descrittore di task.

- Una eventuale `task_join`, effettuata su un task già terminato il cui descrittore è stato scartato dalla primitiva `task_create`, si preoccupa di reinserire di nuovo il descrittore all'interno della coda `freedesc`.

Questo modo di procedere, seppur intricato, permette ai Moduli di Schedulazione di rimanere indipendenti da eventuali connessioni tra la primitiva di join e la schedulazione effettuata da tali Moduli.

## 10.10 Delay, Sleep, Pause e Nanosleep

Il Kernel Generico mette a disposizone più primitive per implementare la autosospensione di un task. Di seguito sono indicate tali funzione e la differenza tra di esse:

`task_delay` Questa primitiva, implementata nel file `kernel/sleep.c`, non fa parte dello standard POSIX e rappresenta la sospensione volontaria di un task per almeno un determinato numero di microsecondi, passati come parametro. Inoltre la primitiva NON è un punto di cancellaazione;

`sleep` Questa funzione autosospende il tak in esecuzione per un determinato numero di secondi. Il task può anche venire risvegliato da un delivery del segnale;

`pause` Questa funzione sospende il task fino alla ricezione di un segnale;

`nanosleep` Questa funzione sospende il task in esecuzione per un tempo minimo indicato come parametro di tipo timespec. Il task può anche essere risvegliato dall'arrivo di un segnale, nel qual caso viene restituito il tempo residuo di pausa non effettuata.

Risulta poi importante non confondere la primitiva `task_sleep` con le precedenti. Quest'ultima non fa parte dello standard POSIX e viene utilizzata per segnalare la fine di un'istanza di un task.

## 10.11 Mutex e variabili condition

Il Kernel Generico mette a disposizione dell'utente una serie di funzioni che sono simili, come interfaccia (a parte il nome e poco altro), alle corrispondenti funzioni dello standard POSIX per la gestione dei mutex e delle variabili condition.

Le estensioni apportate all'interfaccia dei Moduli di Gestione delle Risorse descritte nella sezione precedente vengono utilizzate da tali primitive per gestire i vari protocolli di accesso a risorse condivise in modo trasparente.

In particolare, le interfacce delle primitive sono le seguenti (per una descrizione del loro funzionamento si rimanda allo standard POSIX):

```
int mutex_init(mutex_t *mutex, const mutexattr_t *attr);
```

Questa primitiva può essere utilizzata per inizializzare un descrittore di mutex. A differenza della corrispondente funzione POSIX, il parametro `attr` DEVE essere un attributo correttamente inizializzato. Non può assumere quindi il valore `NULL` per indicare un mutex di default.

```
int mutex_destroy(mutex_t *mutex);
```

Questa primitiva può essere utilizzata per deallocare un mutex contenuto all'interno della strutura dati passata come parametro.

```
int mutex_lock(mutex_t *mutex);
```

Questa primitiva implementa una wait bloccante sul mutex passato come parametro.

```
int mutex_trylock(mutex_t *mutex);
```

Questa primitiva implementa una wait non bloccante sul mutex passato come parametro.

```
int mutex_unlock(mutex_t *mutex);
```

Questa primitiva libera una mutua esclusione precedentemente allocata dallo stesso task con la primitiva mutex_lock.

```
int cond_init(cond_t *cond);
```

Questa primitiva è utilizzata per inizializzare una variabile condition memorizzata all'interno della struttura dati cond.

```
int cond_destroy(cond_t *cond);
```

Questa primitiva è utilizzata per deallocare una variabile condition memorizzata all'interno della struttura dati cond.

```
int cond_signal(cond_t *cond);
```

Questa primitiva effettua una segnalazione singola sulla variabile condition passata come parametro. Viene sbloccato al più un solo task.

```
int cond_broadcast(cond_t *cond);
```

Questa primitiva effettua una segnalazione generale sulla variabile condition passata come parametro. Vengono sbloccati tutti i task bloccati sulla variabile condition.

```
int cond_wait(cond_t *cond, mutex_t *mutex);

int cond_timedwait(cond_t *cond, mutex_t *mutex,
                   const struct timespec *abstime);
```

Il task che esegue una di queste primitive viene bloccato ed il mutex passato come parametro viene rilasciato per essere riacquisito al momento del risveglio. Esistono due versioni della primitiva, di cui una ha un timeout per limitare i tempi di bloccaggio. È utile ricordare il fatto che queste funzioni sono punti di cancellazione, come specificato dallo standard POSIX. Pertanto se viene generata una richiesta di cancellazione per un task bloccato su queste funzioni, il task effettivamente termina, ma solo *dopo* aver riacquisito il mutex. Questo fatto implica che ogni chiamata deve essere protetta tramite delle appropriate funzioni di cleanup che si preoccupino di liberare in modo corretto il mutex.

## 10.12 Altre Primitive

In questa sezione verranno elencate alcune primitive fornite dal Kernel Generico. Esse sono implementate all'interno dei file contenuti nella directory kernel.

```
void task_endcycle(void);

void task_sleep(void);
```

Queste primitive hanno l'effetto di terminare l'istanza corrente del task (vedi Sezione 8.1).

```
void task_abort(void);
```

Questa primitiva ha l'effetto di far terminare il task.

`void group_kill(WORD g);`

Questa primitiva uccide tutti i task appartenenti ad un determinato gruppo.

`TIME sys_time(struct timespec *t);`

Questa primitiva può essere utilizzata all'interno delle applicazioni per leggere il tempo di sistema. Il suo funzionamento è identico ad una `ll_gettime` eseguita ad interruzioni disabilitate.

# Chapter 11

# Esempi

In questo capitolo saranno presentati tre esempi che mostreranno la metodologia con cui possono essere scritti sia i Moduli di Schedulazione per implementare algoritmi di schedulazione e server aperiodici, sia i Moduli di Gestione delle Risorse.

Gli esempi si appoggiano direttamente alle relative implementazioni dei Moduli presentati, quindi è utile fare riferimento ai file citati per avere la corrispondente codifica in Linguaggio C.

## 11.1 Modulo di Schedulazione EDF (Earliest Deadline First)

In questa Sezione verrà descritta una possibile implementazione di un Modulo EDF che possiede le seguenti caratteristiche:

- Supporto per task hard periodici e sporadici;
- Garanzia dei task utilizzando il paradigma del fattore di utilizzazione;
- Controllo del tempo consumato dai task mediante l'utilizzo del flag `CONTROL_CAP` per evitare che un task consumi più di quanto dichiarato

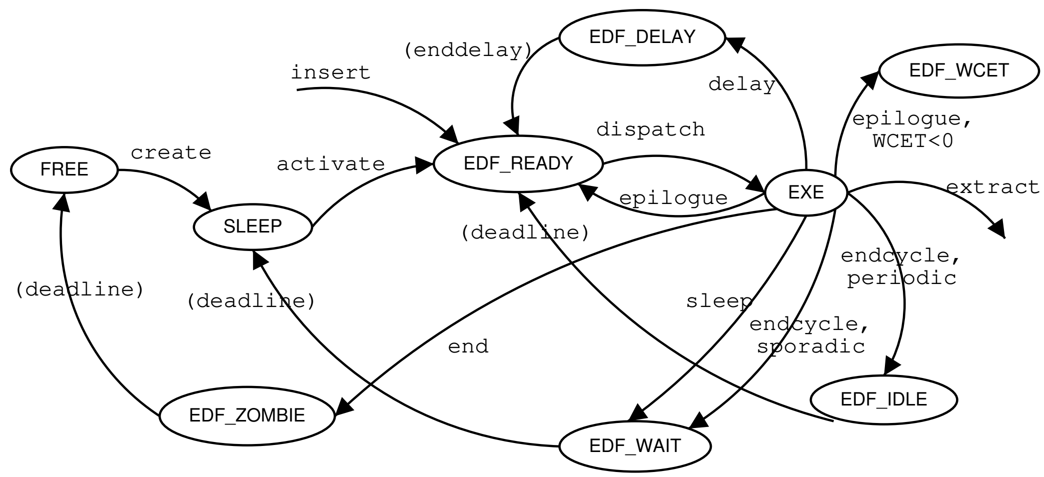


Figure 11.1: Diagramma di transizione degli stati del Modulo EDF.

nel Modello di task.

Tipicamente questo Modulo è posizionato sul Livello 0, in quanto la Garanzia suppone di avere a disposizione tutta la banda del sistema. Per schedulare task periodici in background rispetto ad altri Moduli si consiglia di utilizzare un Modello di task soft, nonchè Moduli di Schedulazione come il CBS. Il Modulo descritto in questa sezione si trova codificato nei file `kernel/modules/edf.c` e `include/modules/edf.h`.

### 11.1.1 Diagramma di transizione degli stati

In Figura 11.1 è descritto il diagramma di transizione degli stati che un task subisce durante la sua vita.

Gli stati il cui nome inizia per `EDF_` sono stati definiti internamente al Modulo. Gli eventi sono indicati vicino alle relative transizioni, tra parentesi sono indicati gli eventi interni al Modulo, generati con l'ausilio degli eventi temporali della OS Lib.

Il significato degli stati introdotti dal Modulo è il seguente:

`EDF_READY` Questo stato è il classico stato pronto in cui si trova un task in attesa di diventare il task con deadline minore.

`EDF_DELAY` Questo stato viene assunto dal task nel momento in cui viene eseguita una primitiva `task_delay`.

`EDF_WCET_VIOLATED` Questo è lo stato in cui finisce un task se esso tenta di consumare più banda di quanto dichiarato. Quando un task finisce in questo stato viene sollevata una eccezione. Sarà poi compito della applicazione considerare eventuali riattivazioni del task in un secondo momento.

`EDF_WAIT` Questo è lo stato in cui finisce un task sporadico alla fine di una istanza. Un task sporadico rimane in questo stato fino all'arrivo della deadline[1]; eventuali attivazioni in questo stato provocano il sollevamento di una eccezione. All'arrivo della deadline il task viene messo nello stato `SLEEP`.

`EDF_IDLE` Questo è lo stato in cui finisce un task periodico alla fine della propria istanza in attesa della riattivazione che avverrà a fine periodo.

`EDF_ZOMBIE` Questo è lo stato in cui finisce un task terminato correttamente. La deallocazione della banda utilizzata dal task viene effettuata all'arrivo della deadline.

### 11.1.2 Strutture dati private

Il Modulo EDF ridefinisce la struttura `level_des` contenente l'interfaccia esportata da un generico Modulo di Schedulazione. Le strutture dati private aggiunte dal Modulo all'interfaccia sono le seguenti:

[1] Si ricorda che i task sporadici vengono garantiti basandosi sul loro minimo tempo di interarrivo.

`period` Questo vettore è utilizzato dal Modulo per memorizzare i periodi (o i minimi tempi di interrarrivo, se un task è sporadico) dei task appartenenti al sistema o inseriti in esso tramite le guest calls.

`deadline_timer` Questo vettore memorizza gli eventi di deadline generati all'interno del Modulo. Per ogni task gestito dal Modulo viene infatti creato un evento di deadline per segnalare riattivazioni ed eventi eccezionali.

`flag` Questo vettore viene utilizzato per memorizzare informazioni interne relative ai task. In particolare sono definiti due flag:

- `EDF_FLAG_SPORADIC` Questo flag è utilizzato per indicare se un task gestito dal sistema è di tipo sporadico.
- `EDF_FLAG_NORAISEEXC` Questo flag è utilizzato per sapere se il livello deve generare un evento di deadline per il task inserito mediante le Guest Calls[2].

`ready` Questa variabile di tipo `QUEUE` è utilizzata per gestire la coda `ready`.

`flags` Il campo flags è utilizzato per memorizzare alcune informazioni passate al momento della registrazione del livello, quali ad esempio eventuali abilitazioni della Garanzia o del controllo del `wcet`.

`U` Questo campo è utilizzato per mantenere una indicazione della banda riservata ai task gestiti direttamente dal Modulo.

Il Modulo inoltre gestisce le capacità utilizzando i campi del descrittore di task messi a disposizione dal Kernel Generico.

[2] Il Modulo EDF accetta come Modelli per i guest task solo dei `JOB_TASK_MODEL`, che hanno un flag per dire se un evento di deadline miss deve essere o meno generato dal Modulo.

### 11.1.3 Funzioni interne al Modulo

Il Modulo EDF ha bisogno di definire alcune funzioni utilizzate internamente. Principalmente si tratta delle funzioni chiamate all'esecuzione degli eventi temporali generati nel sistema. Tali funzioni sono dichiarate come `static` e come tali non sono visibili all'esterno del Modulo.

```
char *EDF_status_to_a(WORD status);
```

Questa funzione è utilizzata internamente alla Level Call `level_status` per convertire gli identificatori degli stati interni del sistema da numero a stringa.

```
void EDF_timer_deadline(void *par);
```

Questa funzione implementa l'evento di deadline all'interno del Modulo.

La prima cosa da fare all'interno di un evento di questo tipo è di recuperare le informazioni relative al proprio descrittore di Modulo. Per fare questo si utilizza un piccolo trucco: il parametro passato nell'evento è in realtà il `PID` del task a cui la deadline si riferisce. Attraverso le seguenti due dichiarazioni diventa così possibile recuperare tutte le strutture necessarie all'esecuzione della funzione:

```
PID p = (PID) par;
EDF_level_des *lev =
    (EDF_level_des *)level_table[proc_table[p].task_level];
```

Questo modo di operare è utilizzato praticamente in quasi tutte le funzioni dei Moduli del Kernel. Ritrovati tutti i parametri la funzione esegue le azioni previste dall'evento di deadline, ovvero:

- Se lo stato del task è `EDF_ZOMBIE` il task viene inserito all'interno della coda `freedesc` e la banda per lui allocata viene liberata;

- Se lo stato è `EDF_IDLE` il task periodico viene riattivato con azioni simili a quelle compiute nella funzione `task_activate`. In particolare viene calcolata la nuova deadline per il task, rigenerato l'evento di deadline nonchè inserito il task all'interno della coda pronti. Infine viene richiesta la rischedulazione del sistema in quanto un nuovo task è divenuto pronto.

- Se lo stato è `EDF_WAIT` il task viene riportato nello stato `SLEEP` in modo tale da poter nuovamente essere riattivato.

- Se il task non si trova in nessuno degli stati precedenti viene generata una eccezione di deadline miss.

```
void EDF_timer_guest_deadline(void *par);
```

Questo evento è generato per i task, inseriti nel modulo tramite le guest calls, che richiedono nel Modello di task la generazione dell'eccezione di deadline miss. L'unica azione compiuta dall'evento è quella di generare tale evento.

```
void EDF_timer_delay(void *par);
```

Questo evento è generato per indicare la terminazione di una primitiva di `task_delay`. Le azioni compiute dall'evento sono quelle di risvegliare il task reinserendolo all'interno della coda pronti e di richiedendo la rischedulazione del sistema.

### 11.1.4 Level Calls

Il Modulo si preoccupa di ridefinire nel modo corretto le level calls dell'interfaccia. Di seguito viene descritta l'implementazione di tali funzioni.

Tutte le funzioni di interfaccia ricevono un parametro di tipo `LEVEL` che può essere utilizzato in modo simile al parametro passato alle funzioni di evento per recuperare tutte le informazioni utili quali il descrittore di Modulo.

```
int EDF_level_accept_task_model(LEVEL l, TASK_MODEL *m);
```

La funzione si preoccupa di controllare se il Modello passato è un Modello di Task gestibile. Il Modulo gestisce solamente Modelli di task Hard (controllando in modo adeguato il campo `pclass`) che hanno un wcet ed un tempo minimo di interarrivo diversi da 0.

```
int EDF_level_accept_guest_model(LEVEL l, TASK_MODEL *m);
```

La funzione si preoccupa di controllare se il Modello passato è un Modello di Task gestibile. Il Modulo gestisce solamente Modelli di task di tipo Job (controllando in modo adeguato il campo `pclass`).

```
void EDF_level_status(LEVEL l);
```

Questa funzione si occupa di stampare in un formato leggibile il contenuto della coda `ready` e di eventuali task appartenenti al Modulo non presenti in tale coda.

```
PID EDF_level_scheduler(LEVEL l);
```

Lo schedulatore del modulo, come ogni buon schedulatore che si rispetti, si limita a ritornare il primo task in coda `ready`.

```
int EDF_level_guarantee(LEVEL l, bandwidth_t *freebandwidth);
```

La funzione di garanzia del Modulo essenzialmente verifica se esiste sufficiente banda libera per schedulare i propri task. In caso positivo decrementa la banda libera della banda effettivamente riservata per i task di un modulo ed indica che il task set può essere garantito.

Se la garanzia è richiamata in seguito ad una creazione di un task all'interno del Modulo, può verificarsi la situazione in cui il nuovo task, assieme a quelli già garantiti dal solo Modulo EDF, viene ad occupare una banda che è maggiore della banda unitaria (e pertanto non rappresentabile all'interno della variabile U[3]). In questo caso, all'interno della funzione `task_create` viene settato un flag che impone il fallimento della garanzia.

### 11.1.5 Task Calls

```
int EDF_task_create(LEVEL l, PID p, TASK_MODEL *m);
```

Questa funzione si occupa di impostare i campi `period`, `flag`, e `wcet` del task appena creato. La funzione setta inoltre il flag `CONTROL_CAP` per indicare al Kernel Generico di controllare il tempo di esecuzione del task. Si preoccupa inoltre di allocare la banda del sistema, in modo tale da poterla controllare nella procedura di garanzia. Se la banda allocata ai task di EDF supera con il nuovo task l'unità, viene settato un flag per indicare che la procedura di garanzia deve fallire.

```
void EDF_task_detach(LEVEL l, PID p);
```

Questa funzione si preoccupa di recuperare la banda eventualmente allocata nella `task_create`, disabilitando se necessario il flag settato nella primitiva `task_create` in caso di garanzia impossibile.

```
int EDF_task_eligible(LEVEL l, PID p);
```

La funzione ristorna sempre `0` per indicare che i task del Modulo EDF sono sempre eleggibili. Il Modulo EDF infatti non utilizza le guest functions di un altro Modulo per gestire i propri task; la `task_eligible` viene quindi richiamata solo sui task ritornati dallo scheduler dello stesso Modulo, che ovviamente sono schedulabili.

[3] Il tipo `bandwidth_t` contiene un numero normalizzato compreso tra 0 ed 1.

```
void EDF_task_dispatch(LEVEL l, PID p, int nostop);
```

Per effettuare il dispatch di un task è sufficiente rimuovere tale task dalla coda pronti, affinchè tale task possa essere messo in esecuzione.

```
void EDF_task_epilogue(LEVEL l, PID p);
```

La funzione ha il compito di implementare la sospensione del task in esecuzione a causa di una preemption o di una fine capacità. Pertanto, la prima cosa che viene fatta nella funzione è quella di controllare se la capacità è terminata. In tal caso, viene sollevata una eccezione ed il task viene posto nello stato `EDF_WCET_VIOLATED`[4].

Nel caso in cui il task ha ancora della capacità da consumare viene invece rimesso in coda pronti.

```
void EDF_task_activate(LEVEL l, PID p);
```

La funzione ha il compito di attivare il task, portandolo quando possibile in coda pronti. In effetti un task può essere attivato solo se si trova nello stato `SLEEP` o `EDF_WCET_VIOLATED`. Nel caso in cui il task si trovi nello stato `EDF_WAIT` significa che l'attivazione del task, che in questo caso può essere solo sporadico, non è corretta, ed una eccezione viene pertanto sollevata.

I compiti svolti dalla funzione sono i seguenti:

- Leggere il tempo corrente mettendolo nel campo `request_time` del descrittore di task;
- Calcolare una deadline per il task;
- Inserire il task attivato in coda pronti;
- Generare l'evento di deadline per il task.

[4]Il task non deve essere estratto da nessuna coda in quanto tale estrazione è stata fatta precedentemente nella funzione `task_dispatch`.

`void EDF_task_insert(LEVEL l, PID p);`

La funzione semplicemente reinserisce in coda pronti un task bloccato su di una sincronizzazione tramite una chiamata alla funzione `task_extract`.

`void EDF_task_extract(LEVEL l, PID p);`

La funzione deve implementare il bloccaggio per sincronizzazione di un task. La funzione non compie alcuna azione, dato che:

- Il task è già stato estratto dalla coda pronti;
- L'evento di capacità è già stato gestito dal Kernel;
- Lo stato deve essere settato dalla primitiva chiamante;
- La deadline non ha bisogno di modifiche.

`void EDF_task_endcycle(LEVEL l, PID p);`

La funzione compie essenzialmente due azioni:

- la ricarica del `wcet` del task, dato che il task ha finito la propria attivazione;
- il settaggio dello stato del task su `EDF_IDLE` o `EDF_WAIT`, in attesa dell'arrivo della deadline.

`void EDF_task_sleep(LEVEL l, PID p);`

La funzione ha il compito di fermare il task. Essa è identica alla primitiva `task_endcycle` eccetto il fatto che per fermare il task si cambia il suo stato direttamente nello stato `EDF_WAIT` in modo da impedire eventuali riattivazioni prima della fine del periodo del task.

```
void EDF_task_end(LEVEL l, PID p);
```

La funzione ha il compito di far terminare il task all'interno del livello. In realtà essa imposta lo stato del task sullo stato `EDF_ZOMBIE`. La deallocazione vera e propria della banda allocata e del descrittore del task viene effettuata all'interno dell'evento che ha come tempo di attivazione la deadline del task.

```
void EDF_task_delay(LEVEL l, PID p, TIME usdelay);
```

Questa funzione ha il compito di sospendere il task per `usdelay` microsecondi. In pratica la funzione si limita a porre il task nello stato `EDF_DELAY` generando un apposito evento di risveglio.

### 11.1.6 Guest Calls

Il Modulo EDF è capace di accettare, oltre al Modello di task Hard gestito direttamente, anche il modello di task Job che viene gestito attraverso le Guest Calls. Notare come il Modello di task accettato come guest non preveda tra i suoi parametri una indicazione della capacità da garantire al task, pertanto il modulo in questione non si preoccupa minimamente di gestirne i tempi di esecuzione (sarà un compito effettuato dal Modulo che inserisce i task come guest). Di seguito è presentata una descrizione di tali funzioni.

```
int EDF_guest_create(LEVEL l, PID p, TASK_MODEL *m);
```

Questa funzione viene chiamata da un generico server aperiodico per inserire un task all'interno del Modulo. La funzione si limita a riempire i campi privati del Modulo con i parametri passati nel Modello di Task. Non viene effettuata alcun tipo di garanzia sui task inseriti (si suppone che una eventuale garanzia sia a carico del Modulo che inserisce tale task come guest).

`void EDF_guest_detach(LEVEL l, PID p);`

Questa funzione è vuota dato che la creazione di un guest task non implica una garanzia del sistema, e dato che non è stata allocata della memoria dinamica.

`void EDF_guest_dispatch(LEVEL l, PID p, int nostop);`

Questa funzione viene tipicamente richiamata dalla chiamata della funzione `task_dispatch` del Modulo che ha inserito il task come guest. L'effetto della funzione è quello di estrarre il task dalla coda pronti (ed è pertanto identico alla `task_dispatch` descritta poco sopra).

`void EDF_guest_epilogue(LEVEL l, PID p);`

Questa funzione è richiamata quando il task in oggetto subisce preemption (su di esso non vengono gestite le capacità...). La funzione inserisce semplicemente il task all'interno della coda `ready`.

`void EDF_guest_activate(LEVEL l, PID p);`

Questa funzione viene richiamata quando il Modulo che ha inserito il task intende attivarlo. L'effetto di questa funzione è quello di inserire il task in coda `ready`, oltre a generare, se richiesto nel Modello passato al momento della creazione, un apposito evento di deadline.

`void EDF_guest_insert(LEVEL l, PID p);`

`void EDF_guest_extract(LEVEL l, PID p);`

`void EDF_guest_delay(LEVEL l, PID p, TIME usdelay);`

Queste funzioni sono implementate in modo del tutto identico alle corrispondenti task calls, anche se il Modello di task Job non le richiederebbe esplicitamente..

```
void EDF_guest_endcycle(LEVEL l, PID p);
```

```
void EDF_guest_sleep(LEVEL l, PID p);
```

Il Modello di task gestito non prevede il bloccaggio o la sospensione di un Job, pertanto queste funzioni sollevano una appropriata eccezione.

```
void EDF_guest_end(LEVEL l, PID p);
```

Questa funzione viene richiamata da un Modulo quando esso intende far terminare un task inserito come guest. Questa funzione non viene richiamata solo alla reale terminazione del task ma anche quando in generale un Modulo deve "cambiare le carte in tavola" (ad esempio, quando occorre posticipare una deadline, o cose simili).

La funzione ha come effetto quello di eliminare ogni traccia del task dalle strutture dati del Modulo. Notare come tale funzione possa essere richiamata anche quando il task in questione non si trova nello stato EXE, quindi occorre controllare tutti i possibili stati in cui esso si può trovare.

### 11.1.7 Funzioni esportate dal Modulo

Il Modulo EDF infine esporta due funzioni che possono essere utilizzate dall'utente.

La prima funzione è la funzione di registrazione, utilizzata per registrare il Modulo all'interno del sistema ed inizializzare le strutture dati interne.

La seconda funzione, `EDF_usedbandwidth`, è utilizzabile per ottenere la banda attualmente allocata dal Modulo. Tale funzione necessita come parametro il livello a cui viene registrato il Modulo EDF, per poter così accedere alle strutture dati private. Questo fatto non è una violazione all'indipendenza della applicazione dai Moduli Registrati, in quanto se l'applicazione intende leggere un particolare dato di un Modulo significa anche che tale applicazione ha anche una conoscenza riguardante la configurazione di tali Moduli.

## 11.2 Modulo di Schedulazione PS (Polling Server)

In questa Sezione verrà descritta una possibile implementazione di un Modulo PS che possiede le seguenti caratteristiche:

- Supporto per task soft aperiodici;
- Garanzia dei task utilizzando il paradigma del fattore di utilizzazione;
- Controllo del tempo consumato dai task senza l'utilizzo del supporto del Kernel Generico (tramite il flag `CONTROL_CAP`);
- Possibilità di utilizzare l'idle time lasciato libero dai Moduli superiori;
- Supporto per le attivazioni pendenti dei task;
- Compatibilità con algoritmi statici (RM) e dinamici (EDF).

Questo Modulo implementa un server aperiodico che inserisce i propri task all'interno di un altro Modulo di Schedulazione, senza avere minimamente idea di come tale Modulo sia implementato. Il Modulo descritto in questa sezione si trova codificato nei files `kernel/modules/ps.c` e `include/modules/ps.h`.

### 11.2.1 Diagramma di transizione degli stati

In Figura 11.2 è mostrato il diagramma di transizione degli stati che un task subisce durante la sua vita.

Il Modulo introduce un solo stato, `PS_WAIT`, in cui parcheggia momentaneamente i task attivi in attesa che arrivi il loro turno di schedulazione (la coda del server è gestita in modo FIFO). Gli altri stati in cui va a passare il task sono di competenza del Modulo Master.

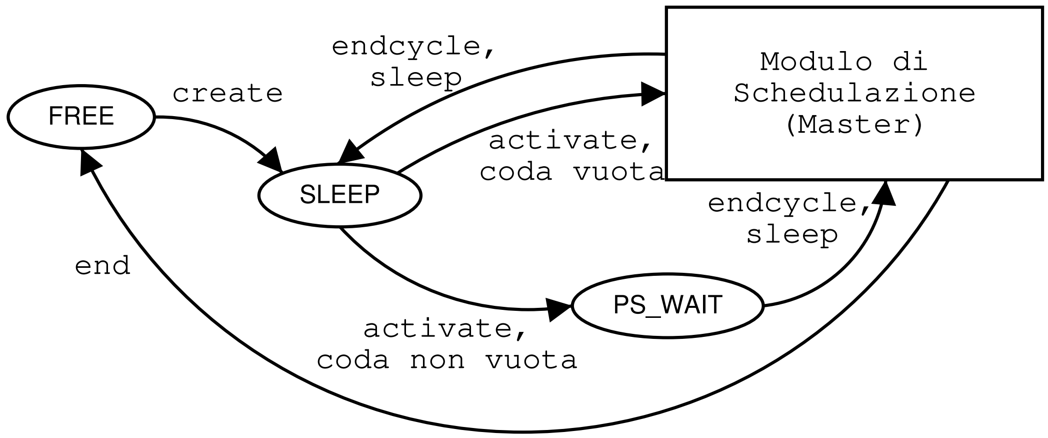


Figure 11.2: Diagramma di transizione degli stati del Modulo PS.

### 11.2.2 Strutture dati private

Il Modulo PS ridefinisce la struttura `level_des` contenente l'interfaccia esportata da un generico Modulo di Schedulazione. Le strutture dati private aggiunte dal Modulo all'interfaccia sono le seguenti:

`nact` Questo vettore è utilizzato per mantenere traccia delle attivazioni pendenti dei task gestiti dal Modulo. Gli elementi del vettore valgono `-1` (se un task non registra le attivazioni pendenti), oppure assumomo un valore `>=0` (ad indicare il numero di attivazioni pendenti del task correlato);

`lastdline` Questo campo è utilizzato per memorizzare la deadline con la quale sta funzionando il Polling Server;

`period` Questo campo contiene il periodo di riattivazione del server.

`Cs` Questo campo contiene la capacità Massima del server;

`availCs` Questo campo contiene il tempo di calcolo disponibile per il server ad un dato istante di tempo. La capacità deve essere aggiornata quando un task gestito dal modulo viene schedulato normalmente dal Modulo Master, e quando un task gestito dal modulo viene indicato tramite il campo shadow da un altro task bloccato a priorità maggiore ed il server non sta schedulando in background.

`wait` Questo campo contiene la coda di attesa in cui sostano le richieste aperiodiche in attesa di essere servite;

`activated` Questo campo indica il PID del task che attualmente è stato inserito all'interno del Modulo di Schedulazione. Il Server inserisce in tale livello non più di un task contemporaneamente.

`flags` Il campo flags è utilizzato per memorizzare alcune informazioni passate al momento della registrazione del livello, quali ad esempio il tipo di garanzia effettivamente implementare (quella di RM o di EDF). Tale campo è inoltre utilizzato per sapere se il server sta o meno eseguendo dei task in background.

`U` Questo campo è utilizzato per mantenere una indicazione della banda riservata al server per i task gestiti direttamente dal Modulo.

`scheduling_level` Questo campo indica il livello in cui è registrato il Modulo di schedulazione utilizzato come Modulo Master.

### 11.2.3 Funzioni interne al Modulo

Il Modulo PS ha bisogno di definire alcune funzioni utilizzate internamente. Principalmente si tratta delle funzioni chiamate all'esecuzione degli eventi temporali generati nel sistema. Tali funzioni sono dichiarate come `static` e come tali non sono visibili all'esterno del Modulo.

`char *PS_status_to_a(WORD status);`

Questa funzione è utilizzata internamente alla Level Call `level_status` per convertire gli identificatori degli stati interni del sistema da numero a stringa.

`void PS_activation(PS_level_des *lev);`

Questa funzione, definita come inline, si occupa di gestire l'attivazione di un task all'interno del livello master. In particolare:

- inizializza un Modello di task Job da passare al livello Master con il periodo e la deadline del Polling Server;
- crea ed attiva tramite le guest calls il task indicato dalla variabile privata lev-`activated`.

`void PS_deadline_timer(void *a);`

Questa funzione implementa la riattivazione periodica del Polling Server. In particolare:

- viene ricalcolata la nuova deadline e rigenerato un nuovo evento di deadline;
- viene ricaricata la capacità del Server al valore massimo se la capacità residua era positiva, ad un valore minore del massimo (dato dalla somma della capacità residua più la ricarica) se negativa;
- se la ricarica è sufficiente a rendere positiva la capacità del server viene attivato un task pendente oppure, in mancanza di task, viene azzerata la capacità appena aggiornata.

### 11.2.4 Level Calls

Il Modulo si preoccupa di ridefinire nel modo corretto le level calls dell'interfaccia. Di seguito viene descritta l'implementazione di tali funzioni.

Tutte le funzioni di interfaccia ricevono un parametro di tipo `LEVEL` che può essere utilizzato in modo simile al parametro passato alle funzioni di evento per recuperare tutte le informazioni utili quali il descrittore di Modulo.

```
int PS_level_accept_task_model(LEVEL l, TASK_MODEL *m);
```

La funzione si preoccupa di controllare se il Modello passato è un Modello di Task gestibile. Il Modulo gestisce solamente Modelli di task Soft aperiodici ignorando gli eventuali parametri aggiuntivi del Modello di task passato.

```
int PS_level_accept_guest_model(LEVEL l, TASK_MODEL *m);
```

Il Modulo non accetta guest task da altri Moduli, pertanto la chiamata restituisce sempre -1.

```
void PS_level_status(LEVEL l);
```

Questa funzione si occupa di stampare in un formato leggibile il contenuto della coda `wait` e dell'eventuale task attualmente inserito all'interno della coda.

```
PID PS_level_scheduler(LEVEL l);
```

Questo schedulatore viene utilizzato dal Modulo nel caso in cui il Modulo non utilizzi l'idle time lasciato dai Moduli di Livello superiore. Esso ritorna sempre `NIL` (ovvero il Modulo non ha task da schedulare).

`PID PS_level_schedulerbackground(LEVEL l);`

Questo schedulatore viene utilizzato dal Modulo nel caso in cui il Modulo utilizzi l'idle time lasciato dai Moduli di Livello superiore. Esso setta un flag nel campo `flags` per ricordarsi che il Modulo sta schedulando in background, dopo di che restituisce il primo task in coda. Lo scheduler viene disattivato nel caso in cui un task schedulato dal server si sia bloccato su di una sincronizzazione (vedi la funzione `PS_task_extract`).

`int PS_level_guaranteeRM(LEVEL l, bandwidth_t *freebandwidth);`

`int PS_level_guaranteeEDF(LEVEL l, bandwidth_t *freebandwidth);`

Queste due funzioni implementano la garanzia all'interno del Modulo, semplicemente decrementando quando possibile la banda libera. La differenza tra le fue funzioni è essenzialmente il fatto che la prima permette di schedulare fino ad una banda totale uguale a `0.69`, mentre la seconda permette di schedulare fino ad una banda limite uguale a `1`. La struttura della funzione è simile a quella della funzione `EDF_level_guarantee`, fatta eccezione del fatto che adesso la garanzia è considerata sul server e non su ogni singolo task creato.

### 11.2.5 Task Calls

`int PS_task_create(LEVEL l, PID p, TASK_MODEL *m);`

Questa funzione si occupa di impostare il campo `nact` per il nuovo task creato in funzione del fatto che il Modello di Task passato specifichi o meno il salvataggio delle attivazioni pendenti. La funzione non setta il flag `CONTROL_CAP` per indicare al Kernel Generico di controllare il tempo di esecuzione del task.

`void PS_task_detach(LEVEL l, PID p);`

Questa funzione non fa nulla, in quanto non sono state allocate nella Task Call `PS_task_create` delle strutture dinamiche, ed in quanto i task del Polling Server non necessitano di garanzia.

`int PS_task_eligible(LEVEL l, PID p);`

La funzione restituisce sempre `0` per indicare che i task del Modulo PS sono sempre eleggibili.

`void PS_task_dispatch(LEVEL l, PID p, int nostop);`

Per effettuare il dispatch di un task è necessario controllare che il task di cui stato richiesto il dispatch sia inserito nella coda wait o sia il task che il Server ha inserito nel Modulo Master.

Nel primo caso il task viene estratto dalla coda `wait`, nel secondo viene richiamata la funzione `guest_dispatch` per indicare al Modulo Master di effettuare il dispatch del task indicato.

Infine verrà generato l'evento di fine capacità se e solo se il parametro `nostop` vale `0` (non c'è alcuna sostituzione dovuta al meccanismo degli shadow). L'evento di fine capacità è creato utilizzando la variabile `cap_timer` in modo tale che esso venga automaticamente rimosso dal Kernel Generico.

`void PS_task_epilogue(LEVEL l, PID p);`

La funzione ha il compito di implementare la sospensione del task in esecuzione a causa di una preemption o di una fine capacità.

Innanzitutto la funzione aggiorna la capacità del server utilizzando nel modo appropriato le variabili `cap_lasttime` e `schedule_time`. Se a questo punto la funzione rileva che la capacità del server è terminata, il task in esecuzione sul livello Master (se non si sta schedulando in background) viene terminato con una `guest_end`, ed inserito nella coda wait.

Se invece la capacità del server non è terminata, possono verificarsi due casi: se il Modulo stava schedulando in background, il task viene rimesso in testa alla coda wait, altrimenti viene richiamata la `guest_epilogue` per segnalare al Modulo master una preemption del task.

`void PS_task_activate(LEVEL l, PID p);`

La funzione ha il compito di attivare il task. Se il task passato come parametro è il task attualmente inserito nel Modulo Master oppure è inserito nella coda `wait` viene memorizzata (se specificato dal task al momento della sua creazione) una attivazione pendente.

In caso contrario vengono svolte le normali attività svolte dalla primitiva di `activate`:

- viene letto il tempo corrente mettendolo nel campo `request_time` del descrittore di task;
- viene attivato il task tramite la `PS_activation` se non ci sono task nel Modulo e la capacità del server è positiva, altrimenti il task è accodato nella coda wait.

`void PS_task_extract(LEVEL l, PID p);`

La funzione deve implementare il bloccaggio per sincronizzazione di un task. La funzione ha il compito di bloccare l'intero server, richiamando eventualmente la `guest_end` sul livello Master e settando il flag `PS_BACKGROUND_BLOCK` per bloccare una eventuale schedulazione in background.

`void PS_task_insert(LEVEL l, PID p);`

La funzione ha il compito di riattivare un task bloccato. La riattivazione del task consiste nel suo inserimento nella coda `wait` e nel reset del flag settato precedentemente nella `task_extract`.

`void PS_task_endcycle(LEVEL l, PID p);`

La funzione compie essenzialmente le seguenti azioni:

- viene aggiornata la capacità del server (o eventualmente disabilitata la schedulazione in background se essa era attiva);
- il task viene estratto dal livello Master tramite una chiamata alla Guest Call `guest_end` altrimenti viene estratto dalla coda `wait`;
- Se il task ha delle attivazioni pendenti viene reinserito in fondo alla coda `wait`, altrimenti viene messo in stato `SLEEP`.
- Se possibile viene prelevato dalla coda `wait` e riattivato un nuovo task tramite una chiamata alla funzione `PS_activation`;

`void PS_task_sleep(LEVEL l, PID p);`

La funzione si comporta in modo identico alla `task_endcycle`, ad eccezione del fatto che essa azzera incondizionatamente le attivazioni pendenti mettendo il task in stato `SLEEP`.

`void PS_task_end(LEVEL l, PID p);`

La funzione si comporta in modo identico alla `task_endcycle`, ad eccezione del fatto che essa invece di inserire il task in coda `wait` o di settare lo stato del task su `SLEEP` inserisce il task direttamente nella coda `freedesc`.

`void PS_task_delay(LEVEL l, PID p, TIME usdelay);`

Questa funzione ha il compito di sospendere il task per `usdelay` microsecondi. In pratica la funzione si limita ad aggiornare la capacità del server ed a richiamare la funzione `guest_delay` del Modulo Master.

### 11.2.6 Guest Calls

Il Modulo PS non accetta guest tasks, pertanto tutte le Guest Calls sollevano una eccezione.

### 11.2.7 Funzioni esportate dal Modulo

Il Modulo PS infine esporta due funzioni che possono essere utilizzate dall'utente.

La prima funzione è la funzione di registrazione, utilizzata per registrare il Modulo all'interno del sistema ed inizializzare le strutture dati interne. Tale funzione registra anche una funzione di inizializzazione che ha il compito di creare il primo evento di deadline per il server. Tale evento non può essere creato all'interno della funzione di Registrazione in quanto quando essa viene richiamata la OS Lib non è ancora inizializzata (e con essa, le funzioni per la gestione degli eventi).

La seconda funzione, `PS_usedbandwidth`, è utilizzabile per ottenere la banda attualmente allocata dal Modulo, ed è simile come funzionalità alla `EDF_usedbandwidth`.

## 11.3 Modulo che implementa il Protocollo PI (Priority Inheritance)

In questa Sezione verrà descritta una possibile implementazione del protocollo di accesso a risorse condivise Priority Inheritance (PI) [SRjPL90] che possiede le seguenti caratteristiche:

- supporto dell'interfaccia messa a disposizione dal Kernel Generico per la gestione dei mutex;

- utilizzo del meccanismo degli shadow messo a disposizione dal Kernel Generico;

- indipendenza dell'implementazione proposta dalle strutture dati utilizzate internamente dai Moduli di Schedulazione che supportano i task che utilizzano i mutex del protocollo;

- possibilità di inizializzare staticamente i mutex utilizzati.

Questo modulo, come del resto tutti gli altri moduli che implementano protocolli di accesso a risorse condivise, è implementato tramite una estensione della interfaccia esportata dai Moduli di Gestion delle Risorse.

Non esistono particolari raccomandazioni relativamente al livello in cui tale Modulo deve essere registrato. Il Modulo descritto in questa sezione si trova codificato nei file `kernel/modules/pi.c` e `include/modules/pi.h`.

### 11.3.1 Approccio utilizzato

L'implementazione del protocollo di Priority Inheritance è semplificata e non complicata dall'utilizzo del meccanismo degli shadow.

Le idee di base sono le seguenti:

- quando si entra in una sezione critica occupando un mutex viene registrato il task che ottiene la muta esclusione, sia per verificare che sia lo stesso task a rilasciarla, sia per utilizzare tale informazione in caso di bloccaggi;

- il Modulo registra inoltre il numero di mutex acquisite dai task, per controllare che un task non muoia possedendo un mutex;

- quando un task tenta di bloccare una mutua esclusione occupata, il proprio shadow viene impostato in modo tale da farlo puntare verso il task bloccante;

- quando un mutex viene rilasciato, vengono liberati tutti i task bloccati su di essa, che così possono riprovare ad acquisire il semaforo (chiaramente una volta liberati i task, il mutex sarà riacquisito dal task a priorità maggiore).

### 11.3.2 Strutture dati private

Il Modulo PI ridefinisce la struttura `mutex_resource_des` contenente l'interfaccia esportata da un generico Modulo di Schedulazione. Le strutture dati private aggiunte dal Modulo all'interfaccia sono le seguenti:

`nlocked` Questo array contiene il numero di mutue esclusioni correntemente acquisite dai task del sistema;

`blocked` Questo array viene utilizzato per tenere traccia dei task che si sono bloccati su di una determinato mutex. In pratica ogni mutex di tipo Priority Inheritance possiede un puntatore al primo task bloccato; i task successivi si trovano concatenati all'interno di questa struttura dati (vedi Figura 11.3). La struttura dati può essere allocata localmente al modulo (e non localmente al mutex) in quanto un task sarà al più bloccato su di un solo mutex.

Ogni mutex che utilizza il protocollo Priority Inheritance inoltre utilizza una struttura dati interna allocata dinamicamente dichiarata con il tipo di dato `PI_mutex_t`, contenente i seguenti campi:

`owner` Indica se il mutex è o meno libero. Quando il mutex è libero questo campo vale `NIL`, altrimenti assume il valore del task che lo ha occupato;

`nblocked` Indica quanti task sono attualmente bloccati sul mutex;

`firstblocked` Quando il campo `nblocked` è diverso da zero questo campo indica un primo task bloccato sul mutex (i seguenti sono accessibili scorrendo la catena nel campo blocked del descrittore di Modulo descritto poco sopra).

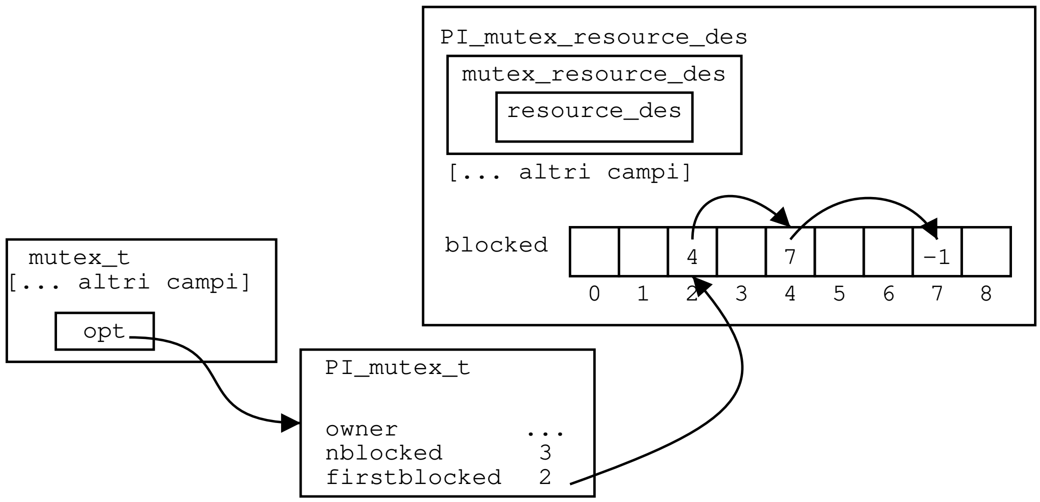


Figure 11.3: Utilizzo dell'array `blocked`. Nell'esempio è descritta a grandi linee la struttura di un `mutex_t` inizializzato con il protocollo Priority Inheritance. Il campo `firstblocked` è il primo elemento della coda dei task bloccati su di un particolare mutex.

Infine, per inizializzare un mutex con il protocollo Priority Inheritance, è necessario passare come parametro alla primitiva `mutex_init` un attributo di tipo opportuno. Tale attributo si chiama `PI_mutexattr_t` e non inserisce alcun campo aggiuntivo rispetto all'attributo di default.

### 11.3.3 Funzioni interne e di interfaccia

```
void PI_resource_status(RLEVEL r);
```

Questa funzione restituisce una indicazione sul numero di mutex attualmente posseduti dai task del sistema. Non vengono mantenute informazioni relative ai mutex Priority Inheritance presenti nel sistema.

`int PI_level_accept_resource_model(RLEVEL l, RES_MODEL *r);`

Questa funzione non accetta alcun Modello di Risorsa, in quanto non necessari per implementare il protocollo[5]. Pertanto, tale funzione restituisce sempre `-1`.

`void PI_res_register(RLEVEL l, PID p, RES_MODEL *r);`

Questa funzione è definita come funzione vuota in quanto essa non viene mai chiamata dal Kernel Generico, non accettando il Modulo dei Modelli di Risorsa.

`void PI_res_detach(RLEVEL l, PID p);`

Questa funzione si limita a controllare che il task che sta terminando non possieda nessun mutex. In caso contrario viene sollevata una eccezione. Una situazione di questo tipo è molto pericolosa in quanto, una volta che il task è definitivamente morto, le strutture degli shadow rimangono inconsistenti (il fatto che un task abbia nel proprio descrittore un campo shadow che punta ad un task inesistente può in generale far andare in crash il sistema)

`int PI_level_accept_mutexattr(RLEVEL l, const mutexattr_t *a);`

Questa funzione si limita a restutire `0` quando un attributo di mutex passato come parametro è di tipo Priority Inheritance.

`int PI_init(RLEVEL l, mutex_t *m, const mutexattr_t *a);`

Questa funzione inizializza un mutex per essere utilizzato con il protocollo implementato. In particolare, viene allocata una struttura dati privata di tipo `PI_mutex_t` e vengono inizializzati tutti i campi delle strutture dati.

[5] A differenza del modulo Priority Inheritance, i Moduli Priority Ceiling (implementato nei files `include/modules/pc.h` e `kernel/modules/pc.c`) ed Stack Resource Policy (implementato nei files `include/modules/srp.h` e `kernel/modules/srp.c`) utilizzano dei Modelli di Risorsa.

```
int PI_destroy(RLEVEL l, mutex_t *m);
```

Questa funzione ha il compito di distruggere un mutex. Il mutex deve essere stato inizializzato correttamente e non deve essere stato occupato da nessun task.

```
int PI_lock(RLEVEL l, mutex_t *m);
```

Questa procedura deve implementare l'occupazione del mutex da parte di un task. Come prima cosa viene verificato se il mutex è stato inizializzato in modo statico. Nel tal caso, viene terminata l'inizializzazione delle strutture dati.

A questo punto, dopo aver controllato che il task che chiama la primitiva non possieda già il mutex da bloccare, viene effettuato un ciclo in cui si attende che la mutua esclusione venga liberata. Il corpo del ciclo semplicemente imposta il campo shadow e rischedula il sistema. Il ciclo è necessario in quanto, quando il mutex verrà rilasciato, tutti i task bloccati su di esso verranno risvegliati entrando in competizione per la sua riacquisizione a seconda delle loro priorità.

Una volta trovato libero, il mutex richiesto viene occupato.

```
int PI_trylock(RLEVEL l, mutex_t *m);
```

Questa funzione è simile alla precedente ad eccezione del fatto che il task non viene mai bloccato in attesa che un mutex venga liberato.

```
int PI_unlock(RLEVEL l, mutex_t *m);
```

Questa funzione ha il compito di rilasciare un mutex. Per prima cosa viene controllato che il task che richiama la primitiva effettivamente possiede tale mutex.

Verificato questo viene rilasciato il mutex e vengono risvegliati tutti i task che si erano bloccati su di esso nel frattempo. Fatto ciò, il sistema viene rischedulato.

```
void PI_register_module(void);
```

Questa funzione si occupa di registrare il modulo all'interno del sistema. La struttura di questa funzione è identica alle analoghe presenti nei Moduli di Schedulazione.

# Capitolo 12

# Implementazione dello standard POSIX

Uno dei maggiori rischi in cui si incorre quando si progettano nuove architetture per Kernel o Sistemi Operativi è quello di dover ricreare ogni volta da zero l'interfaccia che l'utente deve utilizzare per comunicare con il sistema. Il problema non è tanto nella definizione dell'interfaccia stessa (che può essere più o meno completa a seconda degli obiettivi che il sistema si pone), ma nel fatto che non si può pretendere che un utente debba imparare una nuova interfaccia ogni volta che intende utilizzare un nuovo sistema operativo.

Proprio per questo motivo in ambito industriale si è imposto ormai da tempo uno standard, lo standard POSIX [Soc96] [Soc98] che riprende buona parte delle primitive comuni presenti su vari dialetti del Sistema Operativo Unix. La progettazione di un nuovo sistema deve tenere conto di questo standard come riferimento per la interfaccia tra sistema operativo ed applicazioni.

In questo capitolo verrà descritto l'approccio seguito nel supportare l'interfaccia consigliata dallo standard POSIX all'interno dell'architettura modulare proposta per il Kernel Generico.

## 12.1 Funzioni supportate

La definizione completa dello standard non è adatta per essere implementata su sistemi real time dedicati, che spesso per svariati motivi non dispongono di alcune periferiche (quali, ad esempio, le memorie di massa ed i terminali) previste invece dallo standard.

Proprio per questo motivo sono stati proposti e standardizzati dei profili applicativi che intendono delimitare una serie di funzioni necessarie e sufficienti per realizzare software compatibile con lo standard, funzionante in ambienti dedicati per applicazioni realtime.

Tali profili sono contenute nello standard POSIX 1003.13 e sono i seguenti:

**PSE51** Minimal Realtime System Profile

Tipicamente questo tipo di sistemi sono sistemi dedicati al controllo di apparecchiature o dispositivi di I/O. Non è richiesto un file system. Non è richiesta la protezione della memoria. L'ambiente di riferimento è un ambiente monoprocesso multithread.

**PSE52** Realtime Controller System Profile

Questi sistemi sono una estensione del profilo PSE51 e comprendono fondamentalmente un file-system e delle primitive di I/O asincrono.

**PSE53** Dedicated Realtime System Profile

Questi sistemi sono una estensione del profilo PSE51 e prevedono un supporto per la protezione della memoria nonchè un ambiente multiprocesso multithread.

**PSE54** Multi-Pourpose Realtime System Profile

Questi sistemi sono i più "completi" e comprendono tutte le carratteristiche dei precedenti profili. In pratica devono fornire tutte le interfacce fornite dai vari standard POSIX.

I profili che più si avvicinano all'architettura proposta per il Kernel Generico sono i primi due, a seconda che si consideri o meno il supporto per il file system. Questi profili infatti consentono la realizzazione di ambienti applicativi monoprocesso multithread senza protezione della memoria, come quello esportato dalla OS Lib.

Il Kernel non supporta tutte le primitive previste dallo standard POSIX. In particolare esso non implementa le primitive dello standard che riguardano la gestione dei processi, dei terminali e la gestione della localizzazione. Di seguito è presentato un elenco delle funzioni e delle costanti supportate, ordinate a seconda del file header in cui compaiono. La lista non contiene tutti i simboli ma solo le differenze con i simboli elencati all'interno dello standard, in particolare nell'*Annex C (informative) - Header Contents Samples*, pagine 587 e seguenti[1]. Le funzioni riguardanti l'accesso al File System non sono descritte in questo documento ma all'interno di [Gio00].

`aio.h` Non implementato.

`assert.h` Implementato completamente.

`ctype.h` Implementato completamente.

`dirent.h` Non implementato.

`errno.h` Implementato completamente.

`fcntl.h` Implementato completamente.

`float.h` Implementato completamente.

`grp.h` Non implementato.

`limits.h` Implementato completamente.

`locale.h` Non implementato.

[1] La lista seguente comprende circa il 95% delle funzioni previste dal profilo PSE52.

`math.h` Implementato completamente.

`mqueue.h` Implementato completamente[2].

`pthread.h` Implementato parzialmente; in particolare non sono implementate le primitive

- `pthread_atfork`;
- `pthread_condattr_getpshared`;
- `pthread_condattr_setpshared`;
- `pthread_mutexattr_getpshared`;
- `pthread_mutexattr_setpshared`.

Inoltre le macro `PTHREAD_PROCESS_PRIVATE` e `PTHREAD_PROCESS_SHARED` non sono definite . Molte delle funzioni della libreria pthread sono definite inline, spesso richiamando le analoghe funzioni messe a disposizione dal Kernel Generico (vedi Tabella 12.2).

`pwd.h` Non implementato.

`sched.h` Implementato parzialmente; in particolare sono sono implementate le funzioni `sched_getparam`, `sched_getscheduler`, `sched_setparam`, `sched_setscheduler`.

`semaphore.h` Implementato completamente, con l'aggiunta di due primitive non standard, `sem_xpost` e `sem_xwait`.

`setjmp.h` Non implementato.

[2]Le funzioni sono state implementate senza l'utilizzo del filesystem. Pertanto i nomi di file forniti nelle primitive mq_open e mq_unlink vengono utilizzati internamente dal Kernel e non hanno una corrispondenza con files presenti nel filesystem.

`signal.h` Implementato parzialmente; in particolare non sono definite le macro `SA_NOCLDSTOP`, `SIGBUS`, `SIGCHLD`, `SIGCONT`, `SIGHUP`, `SIGINT`, `SIGPIPE`, `SIGQUIT`, `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`; inoltre non è definito il tipo `sig_atomic_t`. Il comportamento dei segnali è identico a quello dello standard, con la sola eccezione che i segnali `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGKILL`, `SIGSEGV` e `SIGTERM` sono presenti come nome ma non vengono mai generati all'interno del sistema. Nei casi di errori irrecuperabili tipicamente la OS Lib termina il sistema senza possibilità di compiere altre azioni.

`stdarg.h` Implementato completamente.

`stddef.h` Implementato completamente.

`stdio.h` Implementato completamente [Gio00].

`stdlib.h` Implementato parzialmente; in particolare non è implementata la funzione `system`. Eventuali ulteriori funzioni non implementate sono citate in [Abe00].

`string.h` Implementato completamente.

`sys/mman.h` Non implementato.

`sys/stat.h` Non implementato.

`sys/times.h` Non implementato.

`sys/types.h` Implementato completamente.

`sys/utsname.h` Implementato completamente.

`sys/wait.h` Non implementato.

`termios.h` Non implementato.

time.h Implementato parzialmente; in particolare sono implementate le macro CLK_TCK, CLOCK_REALTIME, TIMER_ABSTIME, NULL, i tipi clockid_t, clock_t, size_t, timer_t, time_t, timespec, itimerspec, e le primitive clock_getres, clock_gettime, clock_settime, nanosleep, timer_create, timer_delete, timer_getoverrun, timer_gettime, timer_settime.

unistd.h Implementato parzialmente; in particolare non sono implementate le macro

- STDERR_FILENO;
- STDIN_FILENO;
- STDOUT_FILENO;
- _POSIX_ASYNCHRONOUS_IO;
- _POSIX_ASYNC_IO;
- _POSIX_CHOWN_RESTRICTED;
- _POSIX_FSYNC;
- _POSIX_JOB_CONTROL;
- _POSIX_MAPPED_FILES;
- _POSIX_MEMLOCK;
- _POSIX_MEMLOCK_RANGE;
- _POSIX_MEMORY_PROTECTION;
- _POSIX_MESSAGE_PASSING;
- _POSIX_NO_TRUNC;
- _POSIX_PRIORITIZED_IO;
- _POSIX_PRIORITY_SCHEDULING;
- _POSIX_PRIO_IO;

- _POSIX_SAVED_IDS;
- _POSIX_SHARED_MEMORY_OBJECTS;
- _POSIX_SYNCHRONIZED_IO;
- _POSIX_SYNC_IO;
- _POSIX_THREAD_PROCESS_SHARED;
- _POSIX_THREAD_SAFE_FUNCTIONS;
- _POSIX_VDISABLE.

Delle funzioni presenti dichiarate nello standard sono presenti solo `alarm`, `pause`, `sleep`, `sysconf`.

**utime.h** Non implementato.

## 12.2 Implementazione delle funzionalità

Lo standard POSIX non impone alcuna architettura ai Kernel che intendono implementare le primitive proposte. In altre parole, POSIX fornisce solo la interfaccia delle primitive da supportare, senza fornire vincoli sulla loro implementazione.

In ogni caso tutte le implementazioni dello standard seguono alcune linee guida che trapelano in parte anche dai Rationale. Se si confronta la struttura di un kernel tradizionale, che implementa la interfaccia delle primitive POSIX, con l'architettura proposta per il Kernel Generico si può notare come vi siano alcune parti del sistema che non possono essere modularizzate. Tali parti sono ad esempio la gestione dei segnali, delle funzioni di cancellazione e dei dati thread-specific.

Questo fatto impone alcune scelte architetturali all'interno del Kernel Generico:

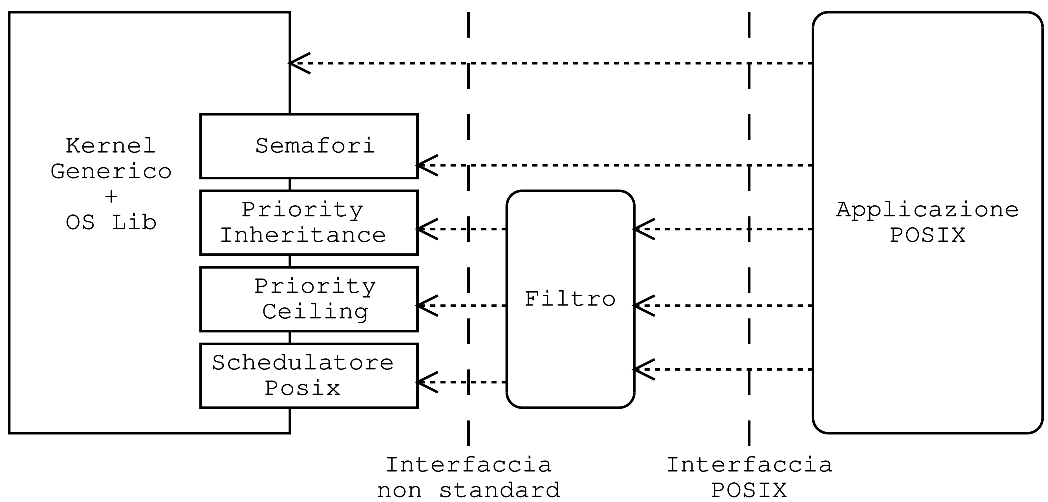


Figura 12.1: Implementazione delle primitive POSIX. L'Applicazione in figura utilizza l'interfaccia POSIX, la quale se necessario rimappa tramite un Filtro le primitive in una serie di chiamate non standard ai Moduli registrati.

- Le parti non modularizzabili della interfaccia POSIX devono venir integrate all'interno dei servizi generali che il Kernel fornisce alle Applicazioni ed ai Moduli;

- Le parti effettivamente modularizzabili, quali il File System, lo schedulatore, i protocolli di accesso a risorse condivise possono invece essere spostate in opportuni Moduli.

Per ottenere un sistema POSIX che supporti le primitive citate sopra occorre quindi registrare opportunamente i moduli corretti all'interno del sistema, definendo alcune primitive POSIX in modo da rimapparle se necessario sui servizi forniti. Uno schema di questo approccio è presentato in Figura 12.1. Questo approccio ha il vantaggio di mantenere separate l'interfaccia proprietaria del Kernel Generico e l'interfaccia dello Standard. L'interfaccia del Kernel Generico può quindi essere utilizzata solo in caso di reale necessità (ad esempio, per sperimentare nuovi algoritmi di schedulazione)[3].

[3]Una altra via possibile (non implementata) poteva essere quella di fornire "di base" una

| Interfaccia POSIX | Interfaccia Non Standard |
|---|---|
| PTHREAD_CANCELED | TASK_CANCELED |
| PTHREAD_CANCEL_ASYNCHRONOUS | TASK_CANCEL_ASYNCHRONOUS |
| PTHREAD_CANCEL_DEFERRED | TASK_CANCEL_DEFERRED |
| PTHREAD_CANCEL_DISABLE | TASK_CANCEL_DISABLE |
| PTHREAD_CANCEL_ENABLE | TASK_CANCEL_ENABLE |
| pthread_t | PID |
| pthread_key_t | task_key_t |
| pthread_cond_t | cond_t |

Tabella 12.1: Simboli del Kernel Generico che svolgono funzioni simili allo standard POSIX.

In Tabella 12.1 sono indicati i simboli dello Standard POSIX implementati direttamente nel Kernel Generico tramite un semplice cambio di nome (ma non di interfaccia). In Tabella 12.2 sono indicate le funzioni dello Standard POSIX implementate direttamente nel Kernel Generico tramite un semplice cambio di nome (ma non di interfaccia).

interfaccia POSIX modificata per accogliere le peculiarità del Kernel Generico. In tutti e due i casi occorre comunque apportare delle modifiche non standard alla interfaccia.

| Interfaccia POSIX | Interfaccia Non Standard |
|---|---|
| pthread_sigmask | task_sigmask |
| pthread_kill | task_signal |
| pthread_mutex_destroy | mutex_destroy |
| pthread_mutex_lock | mutex_lock |
| pthread_mutex_trylock | mutex_trylock |
| pthread_mutex_unlock | mutex_unlock |
| pthread_cond_destroy | cond_destroy |
| pthread_cond_broadcast | cond_broadcast |
| pthread_cond_signal | cond_signal |
| pthread_cond_wait | cond_wait |
| pthread_cond_timedwait | cond_timedwait |
| pthread_join | task_join |
| pthread_detach | task_unjoinable |
| pthread_exit | task_abort |
| pthread_key_create | task_key_create |
| pthread_getspecific | task_getspecific |
| pthread_setspecific | task_setspecific |
| pthread_key_delete | task_key_delete |
| pthread_setcancelstate | task_setcancelstate |
| pthread_setcanceltype | task_setcanceltype |
| pthread_testcancel | task_testcancel |
| pthread_cleanup_push | task_cleanup_push |
| pthread_cleanup_pop | task_cleanup_pop |

Tabella 12.2: Nomi delle primitive del Kernel Generico che svolgono funzioni simili allo standard POSIX.

# Capitolo 13

# Prestazioni

Le operazioni svolte dal Kernel Generico e i metodi con cui la OS Lib gestisce gli eventi temporali influenzano in modo notevole il comportamento delle applicazioni. Risulta pertanto importante effettuare delle misurazioni che forniscano una idea dell'overhead introdotto dalle sezioni critiche all'interno del sistema[1].

In questo capitolo vengono illustrate alcune misurazioni riguardanti le prestazioni di alcuni aspetti del Kernel Generico ritenuti importanti, quali ad esempio il cambio di contesto e gli overhead dovuti alle varie modalità di gestione degli eventi nella OS Lib.

## 13.1 Metodologia utilizzata

La metodologia utilizzata per eseguire le misurazioni all'interno del Kernel è stata quella di instrumentare opportunamente il codice sorgente in modo tale da leggere il tempo di sistema in alcuni punti ben precisi. Le letture sono state effettuate tramite le apposite funzioni messe a disposizione dalla OS Lib; tali funzioni basano la loro lettura del tempo sulle informazioni

[1]Si ricorda che, dato che la OS Lib non implementa la protezione della memoria, le primitive di sistema vengono implementate tramite la disabilitazione delle interruzioni.

recuperabili dal real-time clock del PC, e pertanto sono realizzate con una precisione di circa un microsecondo. Questa precisione in termini assoluti non è elevata, ma è comunque sufficiente per fornire delle informazioni utili per la comprensione dei meccanismi interni del sistema[2]. Quando i tempi in gioco sono confrontabili con il tempo impiegato dal sistema per leggere il clock, più ripetizioni in sequenza dell'evento da misurare sono state effettuate, fornendo poi come risultato un valore medio.

Una misurazione precisa dell'overhead del Kernel può essere fatta solo assumendo una particolare configurazione dei Moduli all'interno del sistema, in quanto al cambiare delle configurazioni variano gli algoritmi implementati e pertanto variano anche gli overhead introdotti. In poche parole, l'approccio modulare rende impossibile una stima generica ed esatta dei tempi; per questo motivo i test realizzati fanno sempre riferimento a configurazioni minime di volta in volta specificate.

Occorre infine notare che il Kernel utilizzato per il testing non ha subito ottimizzazioni del codice, pertanto le prestazioni citate potranno essere incrementate ulteriormente.

## 13.2 Risultati dei test

Tutti i test sono stati realizzati utilizzando un sistema di elaborazione dotato di processore Intel Celeron a 366 Mhz, 64 Mb di RAM e 512 Kbytes di memoria cache di secondo livello.

[2]Misurazioni più precise possono essere realizzate utilizzando il contatore interno dei processori Pentium nonchè schede hardware dedicate alla misurazione dei tempi di risposta agli interrupt. Tali tecniche non sono state adottate per permettere di creare dei test che, seppur imprecisi, utilizzino solamente il supporto fornito dalla OS Lib.

### 13.2.1 Test 1

Il test è contenuto all'interno del file `examples/kernel/perf1.c`, ed intende fornire una stima dei seguenti parametri:

- Tempo impiegato dalla OS Lib ad effettuare la lettura del tempo;
- Tempo medio per il cambio di contesto.

La configurazione del sistema è caratterizzata da un Modulo EDF registrato sul livello 0 e due moduli Round Robin nei livelli successivi.

Per prima cosa l'applicazione esegue 100 volte la lettura del timer di sistema ad interruzioni disabilitate. La differenza di tempo tra la prima e l'ultima lettura è di circa 285-286 $\mu s$, cioè ogni lettura del timer consuma circa 2.8 $\mu s$. Questo tempo è dovuto, oltre ai tempi impiegati per accedere al controllore del timer, ai tempi necessari per effettuare le conversioni di unità di misura tra tick del real-time clock e microsecondi.

Successivamente l'applicazione esegue un task Round Robin con un tempo di calcolo di circa 5 secondi ed un timeslice molto più elevato (in modo da non far mai scattare l'evento di capacità). Alla seconda istanza tale task viene interrotto da un task periodico EDF la cui unica azione è quella di effettuare per 2501 volte la chiamata alla primitiva `task_endcycle`. Misurando le differenze nei tempi di esecuzione dei due casi si può calcolare l'overhead generato dal task. Ogni istanza del task EDF provoca, per la precisione:

- l'esecuzione di un evento di riattivazione del Modulo EDF;
- una prima rischedulazione del sistema, che porta in esecuzione il task EDF;
- l'esecuzione della primitiva `task_endcycle`;
- una seconda rischedulazione del sistema, che riporta in esecuzione il task RR.

| Conf. | Task RR | | | Task EDF | |
|---|---|---|---|---|---|
| Timer | solo | con task EDF | $\Delta_{RR}$ | totale | max |
| one shot | 5426138 $\mu s$ | 5504558 $\mu s$ | 78420 $\mu s$ | 54107 $\mu s$ | 27 $\mu s$ |
| 300 $\mu s$ | 5538523 $\mu s$ | 5578833 $\mu s$ | 40310 $\mu s$ | 29411 $\mu s$ | 14 $\mu s$ |
| 600 $\mu s$ | 5480442 $\mu s$ | 5519804 $\mu s$ | 39362 $\mu s$ | 29335 $\mu s$ | 15 $\mu s$ |
| 1200 $\mu s$ | 5453209 $\mu s$ | 5492105 $\mu s$ | 38896 $\mu s$ | 29160 $\mu s$ | 15 $\mu s$ |
| 2400 $\mu s$ | 5439002 $\mu s$ | - | 38896 $\mu s$ | - | - |

Tabella 13.1: Tempi di calcolo registrati dal test 1.

I tempi di calcolo indicativi sono indicati in Tabella 13.1. Ovviamente i dati indicati cambiano ad ogni esecuzione dei test; in ogni caso, quelli presentati possono essere utilizzati per effettuare alcuni commenti:

- I numeri indicati nella tabella comprendono al loro interno anche gli overhead necessari per leggere il tempo all'interno del sistema; in ogni caso il codice del Kernel non è stato instrumentato;

- Il tempo di calcolo di una istanza del task EDF comprende la schedulazione del sistema e l'esecuzione della primitiva endcycle, fino alla rischedulazione del sistema;

- Il tempo di calcolo del task RR tiene conto, oltre all'esecuzione del task stesso (colonna 2), anche il tempo di gestione della interruzione, l'evento di rischedulazione ed il ritorno da interruzione (colonna 3). L'overhead totale registrato è contenuto nella quarta colonna (contrassegnata dal simbolo$\Delta_{RR}$), contiene può essere stimato in circa 30 $\mu s$ per il timer one shot e circa 16 $\mu s$ per i time periodici;

- Se si considera il tempo massimo registrato dal sistema per una istanza del task EDF (paragonabile in pratica alla esecuzione della sola primitiva di `task_endcycle`) si può notare come tale valore sia quasi raddoppiato nel caso di utilizzo del time one-shot rispetto al timer periodico. Questo è dovuto al fatto che le funzioni della OS Lib che gestiscono gli eventi devono riprogrammare ogni volta il timer di sistema, cosa non necessaria se viene utilizzato un timer periodico.

- Il fatto che la riprogrammazione del timer sia più onerosa nel caso one-shot è evidenziato anche dal tempo totale di esecuzione del task EDF.
- In ogni caso il timer one-shot risulta più performante se considerato globalmente, in quanto il timer periodico genera spesso interruzioni "inutili". Questo fatto è visibile considerando la prima colonna o la somma dei dati presenti nella terza e nella quarta colonna. I dati dimostrano anche come l'overhead del sistema diminuisce all'aumentare del periodo del timer.
- Aumentare il periodo del timer in un approccio periodico porta ad una diminuzione dell'overhead e ad un aumento della imprecisione temporale di gestione degli eventi, che può portare a malfunzionamenti; in tabella un timer periodico di 2400 $\mu s$ non permette l'esecuzione corretta del task EDF che ha un periodo paragonabile (2 $ms$).

### 13.2.2 Test 2

Il test è contenuto all'interno del file `examples/kernel/perf2.c`, ed intende fornire una stima dei seguenti parametri:

- Tempo impiegato all'interno della gestione degli eventi;
- Numero di eventi generati dalla OS Lib.

La configurazione del sistema è caratterizzata da un Modulo EDF registrato sul livello 0, un Modulo CBS sul livello 1, un Modulo TBS sul livello 2 ed un Modulo RR sul livello 3.

L'applicazione eseguita prevede un mix di 5 task periodici con tempo di calcolo "stabile" (circa 6 $\mu s$), un insieme di task CBS (circa 20) che nascono

| Timer | Tempo totale | Tempo massimo | Tempo medio | Lifetime |
|---|---|---|---|---|
| one shot | 176917 $\mu s$ | 37 $\mu s$ | 19.6 $\mu s$ | 38.32 sec |
| 300 $\mu s$ | 30353 $\mu s$ | 27 $\mu s$ | 3.37 $\mu s$ | 2.7 sec |

Tabella 13.2: Tempi di calcolo registrati dal test 2.

e terminano in modo dinamico tentando di saturare la banda libera (i tempi di calcolo sono casuali con media intorno ai 3.5 $\mu s$), e un insieme di 8 task attivati in modo casuale alla terminazione di ogni istanza dei task CBS, con tempo di calcolo di circa 13 $\mu s$.

Viene registrato il tempo trascorso dalla chiamata del prologo a quella dell'epilogo degli eventi della OS Lib dei primi 9000 eventi del sistema instrumentando il codice con opportune chiamate alla funzione `ll_gettime` (il tempo di esecuzione della funzione non è stato scorporato dai dati presentati). I dati rilevati sono presentati in Tabella 13.2. I dati indicati ovviamente cambiano ad ogni esecuzione dei test; in ogni caso, quelli presentati possono essere utilizzati per effettuare alcuni commenti:

- Il timer one shot comporta un maggiore overhead nella gestione degli eventi temporali, che si ripercuote sul tempo di esecuzione totale e massimo dei primi 9000 eventi;

- Gli eventi gestiti dal timer one shot sono generalmente tutti "utili";

- Il tempo medio di esecuzione del timer one shot è dello stesso ordine di grandezza del tempo massimo, mentre il tempo medio di esecuzione degli eventi periodici è sensibilmente più basso. Considerando che i dati comprendono anche il tempo usato dalla funzione di lettura del tempo (circa 2.8 $\mu s$), si può capire come buona parte degli eventi periodici non eseguono alcun evento (sono "inutili");

- Il numero di eventi "inutili" è elevato, infatti se si considera l'intervallo di tempo in cui vengono generati i primi 9000 eventi possiamo notare

| Conf. Timer | Senza altri task Massimo | Medio | N. | Con Applicazione Massimo | Medio | N. |
|---|---|---|---|---|---|---|
| one shot | 16 $\mu s$ | 10.5 $\mu s$ | 382 | 102 $\mu s$ | 12.1 $\mu s$ | 417 |
| 300 $\mu s$ | 303 $\mu s$ | 153.0 $\mu s$ | 379 | 307 $\mu s$ | 154.7 $\mu s$ | 401 |
| 600 $\mu s$ | 601 $\mu s$ | 308.2 $\mu s$ | 373 | 602 $\mu s$ | 295.6 $\mu s$ | 408 |
| 1200 $\mu s$ | 1202 $\mu s$ | 612.0 $\mu s$ | 366 | 1202 $\mu s$ | 594.49 $\mu s$ | 397 |

Tabella 13.3: Tempi di calcolo registrati dal test 3.

come ci sia un ordine di grandezza di differenza tra i due approcci. Aumentare il periodo del timer periodico non aiuta ad aumentare le prestazioni del sistema, in quanto diminuiscono gli eventi inutili ma diminuisce anche la precisione del sistema.

### 13.2.3 Test 3

Il test è contenuto all'interno del file `examples/kernel/perf3.c`, ed intende fornire una stima dei seguenti parametri:

- Ritardo massimo tra l'istante di attivazione di un evento e l'istante di gestione di un evento;

- Ritardo medio tra l'istante di attivazione di un evento e l'istante di gestione di un evento.

La configurazione del sistema è identica a quella del test 2.

Il test si occupa di generare in modo casuale il tempo di attivazione di una serie di eventi. Ogni evento si preoccupa di registrare la differenza tra il proprio tempo di attivazione ed una lettura del timer effettuata all'inizio dell'evento stesso. Inoltre l'evento si preoccupa di rigenerare l'evento successivo (al più un evento di questo tipo è presente nella coda mantenuta dalla OS Lib). I dati rilevati sono presentati in Tabella 13.3. I dati indicati ovviamente cambiano ad ogni esecuzione dei test; in ogni caso, quelli presentati possono essere utilizzati per effettuare alcuni commenti:

- In assenza di carico il timer one shot non subisce ritardi a causa di sezioni critiche o altri eventi in coda, e pertanto i valori ottenuti sono molto bassi e dipendono solo da cause hardware (vedi Figure 5.4 e 5.5);

- In presenza di carico il timer one shot è influenzato molto poco dalla presenza di altri eventi, infatti il ritardo medio è confrontabile con in caso senza carico. Alcune volte si può però verificare che due eventi vengano generati in modo molto ravvicinato: in questo caso il secondo evento viene ritardato di una quantità che dipende essenzialmente dalle caratteristiche dell'hardware (durante il test tale valore era di 100 $\mu s$);

- I ritardi sopportati dagli eventi generati con timer programmato in modo periodico praticamente prescindono dalla presenza di ulteriori eventi in coda. I valori ottenuti sono paragonabili a quelli attesi considerando la quantizzazione del tempo: il tempo massimo è infatti praticamente coincidente con l'intervallo di quantizzazione ed il tempo medio è pari alla metà di tale intervallo.

### 13.2.4 Test 4

Il test è contenuto all'interno del file `examples/kernel/perf4.c`, ed intende fornire una stima riguardante la durata della coppia di primitive di bloccaggio su mutex con protocollo Priority Inheritance.

Il test si occupa di richiamare la coppia di funzioni all'interno di un task per 1000 volte, misurandone il tempo di esecuzione. I risultati di questo test indicano come la coppia di funzioni viene eseguita tipicamente in 12 $\mu s$ utilizzando il timer configurato in modo one shot ed in 5 $\mu s$ utilizzando il timer configurato in modo periodico (i valori rimangono praticamente stabili al variare del periodo di programmazione del timer). La differenza può essere imputata all'overhead introdotto dalle funzioni di programmazione del timer necessarie quando si utilizza il modo one shot.

Per quanto riguarda le altre primitive che prevedono cambi di contesto e rischedulazioni i comportamenti attesi sono simili a quanto riportato per questo test.

# Capitolo 14

# Conclusioni

In questa tesi è stato presentata l'Architettura ed i dettagli del Kernel LEGO, un nuovo Kernel real-time modulare compatibile con lo standard POSIX 1003.13 profilo PSE52.

La chiave di lettura della nuova implementazione è la Modularità allargata a parti dei Kernel tradizionali tipicamente non modulari, quali gli algoritmi di schedulazione e la gestione delle risorse condivise.

La Modularità porta con se notevoli vantaggi, quali l'indipendeza delle Applicazioni dai Moduli registrati a runtime, nonchè l'indipendenza reciproca dei Moduli stessi. Questo permette di utilizzare il Kernel per creare e provare sul campo nuovi algoritmi e strategie di schedulazione, consentendo sia implementazioni tradizionali che modulari dei vari algoritmi presentati in letteratura.

Il raggiungimento degli obiettivi è comunque caratterizzato un trade-off tra i costi, quali overhead ed aumento del codice costituente il Kernel, ed i benefici, citati nei paragrafi precedenti.

Infine la compatibilità del Kernel con lo standard POSIX rende semplice il porting di vecchie applicazioni, nonchè la scrittura di nuove, all'interno del sistema.

Questo documento, assieme al codice sorgente del Kernel, dei Moduli, delle Librerie e degli esempi è materiale liberamente distribuibile come specificato dalla licenza GPL [FSF], ed è distribuito su internet all'indirizzo `http://hartik.sssup.it`.

# Bibliografia


[AB98] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real Time Systems Symposium*, Madrid, Spain, December 1998.

[AB99] Luca Abeni and Giorgio Buttazzo. Constant bandwidth vs proportional share resource allocation. In *Proceedings of the IEEE International Conference on Mutimedia Computing and Systems*, Florence, Italy, June 1999.

[Abe00] Luca Abeni. Library for operating system development. Technical report, Scuola Superiore di Studi e Perfezionamento S. Anna, 2000.

[Bak90] T. P. Baker. A stack-based allocation policy for realtime processes. In *IEEE Real-Time Systems Symposium*, december 1990.

[Bol] Erich Stefan Boleyn. Grub, grand unified bootloader. http://www.gnu.org/software/grub/grub.html.

[BS97] G. C. Buttazzo and F. Sensini. Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments. In *Proceedings of the 3rd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages pp. 39–48, Como, Italy, September, 8-12 1997.

[CBLL98] Antonino Casile, Giorgio Buttazzo, Gerardo Lamastra, and Giuseppe Lipari. Simulation and tracing of hybrid task sets on distributed systems. In *Real Time Computing Systems and Applications*, 1998.

[DW95] Robert Davis and Andy Wellings. Dual Priority Scheduling. In *Proceedings of the IEEE Real Time Systems Symposium*, December 1995.

[EK95] Dawson R. Engler and M. Frans Kaashoek. Exterminate all operating system abstractions. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 78–83, Orcas Island, Washington, May 1995. IEEE Computer Society.

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[FBB+97] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux oskit: A substrate for os and language research. In *16 ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[FS96] Bryan Ford and Sai Susarla. Cpu inheritance scheduling. In *Proceedings of OSDI*, October 1996.

[FSF] Free Software Foundation. GNU General Public License (GPL). http://www.gnu.org/copyleft/gpl.html.

[GB95] T. M. Ghazalie and T.P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9, 1995.

[Gio00] Massimiliano Giorgi. Progetto e realizzazione di un file system modulare. Master's thesis, Università di Pisa, Facoltà di Ingegneria, Corso di laurea in Ingegneria Informatica, Via Diotisalvi, Pisa, May 2000.

[hCN99] Hao hua Chu and Klara Nahrstedt. CPU service classes for multimedia applications. In *Proceedings of the IEEE International Conference on Mutimedia Computing and Systems*, Florence, Italy, June 1999.

[JSMA98] Kevin Jeffay, F.D. SMith, A. Moorthy, and J.H. Anderson. Proportional share scheduling of operating system services for real-time applications. In *IEEE Real Time System Symposium*, Madrid, Spain, December 1998.

[KSD91] K.Jeffay, D.L. Stone, and D.Poirier. Yartos: Kernel suppoprt for efficient, predictable real-time systems. In *IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[LBA98] Giuseppe Lipari, Giorgio Buttazzo, and Luca Abeni. A bandwidth reservation algorithm for multi-application systems. In *IEEE Real Time Computing Systems and Applications*, Hiroshima, Japan, October 1998.

[LES+97] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger. Achieved ipc performance (still the foundation for extensibility). In *Proceedings of the 4th Hot Topics in Operating Systems (HotOS-VI)*, Cape Cod, Massachusetts, May 1997.

[Lie95] Jochen Liedtke. On $\mu$-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, Colorado, December 1995.

[LL73] C. L. Liu and J. Layland. Scheduling alghorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1), 1973.

[LMB+] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. http://www.cl.cam.ac.uk/Research/SRG/netos/pegasus/papers/jsac-jun97/paper.html.

[Nak98] Tatsuo Nakajima. Resource reservation for adaptive qos mapping in real-time mach. In *Sixth International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, April 1998.

[Riv] Wind River. Vxworks 5.4. http://www.wrs.com/products/html/vxwks54.html.

[RJMO98] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[SAWJ+96] Ian Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the IEEE Real Time Systems Symposium*, December 1996.

[SAWJ97] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the duality between resource reservation and proportional share resource allocation. In *SPIE Multimedia Computing and Networking*, volume 3020, pages 207–214, San Jose, CA, February 1997.

[Soc96] IEEE Computer Society, editor. *International Standard ISO/IEC 9945-1: 1996 (E) - IEEE Std 1003.1, 1996 Edition - Information technology - Portable Operating System Interface (POSIX)*. IEEE, 1996.

[Soc98] IEEE Computer Society, editor. *IEEE Standard for Information Technology - Standardized Application Environment Profile - POSIX Realtime Application Support (AEP)*. IEEE, 1998.

[SRjPL90] Lui Sha, Ragunathan Rajkumar, and john P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE transaction on computers*, 39(9), september 1990.

[SSL] QNX Software Systems Ltd. Qnx neutrino real-time os. http://www.qnx.com/products/os/neutrino.html.

[TNR90] H. Tokuda, T. Nakajima, and P. Rao. Real-time mach: Toward a predictable real-time system. In *USENIX Mach Workshop*, pages 73–82, October 1990.

[WL99] Y.C. Wang and K.J. Lin. Implementing a general real-time scheduling framework in the red-linux real-time kernel. In *Proceedings of IEEE Real-Time Systems Symposium*, Phoenix, December 1999.

[WW94] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *First Symposium on Operating System Design and Implementation*, pages 1–12, 1994.